ТРУДЫ

ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN Print 2079-8156
Том 37 Выпуск 6 Часть 1

ISSN Online 2220-6426
Volume 37 Issue 6 Part 1

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2025

ИСП РАН

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

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

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

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

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

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

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

Труды ИСП РАН реферируются и/или индексируются в:

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

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

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

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

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

УДК004.45

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

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

Содержание

О длине адаптивной различающей последовательности для семейства наблюдаемых автоматов.
Бурдонов И.Б., Евтушенко Н.В., Косачев А.С. *7*

Разновидность JavaBeans-компонент: композиция типов из агрегации инстансов.
Гринкруг Е.М. *21*

Сравнение объектно-ориентированного и процедурно-параметрического полиморфизма.
Косов П.В., Легалов А.И. *43*

Статический анализ исходного кода для языка Golang: обзор литературы.
Дворцова В. В., Бородин А.Е. *59*

Повышение точности статического анализа кода при помощи больших языковых моделей.
Панов Д.Д., Шимчик Н.В., Чибисов Д.А., Белеванцев А.А., Игнатьев В.Н. *83*

Предсказание истинности предупреждений промышленного статического анализатора с использованием методов машинного обучения.
Тяжкороб У.В., Беляев М. В., Белеванцев А.А., Игнатьев В.Н. *101*

Быстрые вызовы и раскрытие на месте: гибридная стратегия для встраиваемых функций виртуальных машин.
Заведеев Д.В., Жуйков Р.А., Скворцов Л.В., Пантилимонов М.В. *121*

Аннотирование исходного кода для статического анализа.
Афанасьев В.О., Бородин А.Е., Велесевич Е.А., Орлов Б.В. *135*

Распознавание заголовков таблиц на основе больших языковых моделей.
Охотин И.И., Дородных Н.О. *149*

Улучшение электронных государственных услуг посредством разработки чат-ботов с использованием Azure OpenAI.
Радосављевић Л., Симић М., Јоксимовић А., Наумовић Т., Деспотовић-Зракић М. *167*

Показатели множественного числа существительных в селькупских диалектах.
Ковылин С.В. *181*

Тюрко-монгольские параллели в лексике материальной культуры тюркских языков Урало-Поволжья (на материале названий мастей лошади).
Муратова Р.Т. *193*

Лексика материальной культуры в словаре М. А. Кастрена (1844) и аудиословаре ижемского диалекта говора (2012): сравнительный анализ по данным платформы LingvoDoc.
Баженова О. Н. *203*

Сегментация документов на основе графовых нейронных сетей: от строк к словам.
Копылов Д.Е., Михайлов А.А., Трифонов Р.И. *219*

Сравнение интерпретируемости моделей ResNet50 и ViT-224 в задаче классификации бактерий на снимках сканирующего электронного микроскопа.
Гридин В.Н., Новиков И.А., Салем Б.Р., Солодовников В.И......*233*

Table of Contents

On length of an adaptive distinguishing sequence for a family of observable finite state machines.
Burdonov I.B., Yevtushenko N.V., Kossatchev A.S. *7*

Yet another kind of JavaBeans: composed types from aggregated instances.
Grinkrug E.M. *21*

Comparison of object-oriented and procedural-parametric polymorphism.
Kosov P.V., Legalov A.I. *43*

Static Analysis of Golang Source Code: A Survey.
Dvortsova V.V., Borodin A.E. *59*

Increasing precision of static code analysis using large language models.
Panov D.D., Shimchik N.V., Chibisov D.A., Belevantsev A.A., Ignatyev V.N. *83*

Machine learning-based validation of warnings in an industrial static code analyzer.
Tsiazhkorob U.V., Belyaev M.V., Belevantsev A.A., Ignatiev V.N. *101*

Fast calls and in-place expansion: A hybrid strategy for VM intrinsics.
Zavedeev D.V., Zhuykov R.A., Skvortsov L.V., Pantilimonov M.V. *121*

Source code annotation for static analysis.
Afanasyev V.O., Borodin A.E., Velesevich E.A., Orlov B.V. *135*

Using large language models for table header recognition.
Okhotin I.I., Dorodnykh N.O. *149*

Enhancing e-government services through chatbot development using Azure OpenAI.
Radosavljević L., Simić M., Joksimović A., Naumović T., Despotović-Zrakić M. *167*

Plural number markers of nouns in Selkup dialects.
Kovylin S.V. *181*

Turkic-Mongolian parallels in the vocabulary of the material culture of the Turkic languages of the Ural-Volga region (based on the names of horse colors).
Muratova R.T. *193*

Material Culture Lexicon in M.A.Castrén's Dictionary (1844) and an Audio Dictionary of the Ižma Dialect (2012): Comparative Analysis on LingvoDoc.
Bazhenova O.N. *203*

Segmentation of documents based on graph neural networks: from strings to words.
Kopylov D.E., Mikhailov A.A., Trifonov R.I. *219*

Comparison of the interpretability of ResNet50 and ViT-224 models in the classification task is erroneous on images of a scanned microscope object.
Gridin V.N., Novikov I.A., Salem B.R., Solodovnikov V.I. *233*

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

О длине адаптивной различающей последовательности для семейства наблюдаемых автоматов

[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] *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 статье исследуется длина адаптивной различающей последовательности для семейства инициальных полностью определенных наблюдаемых, возможно, недетерминированных автоматов. Приводится верхняя оценка длины такой последовательности и показывается ее достижимость. Обсуждается, каким образом полученные результаты можно применить при построении адаптивных диагностических тестов на основе конечно-автоматной модели.

Ключевые слова: недетерминированные автоматы; адаптивная различающая последовательность; диагностические тесты.

Для цитирования: Бурдонов И.Б., Евтушенко Н.В., Косачев А.С. О длине адаптивной различающей последовательности для семейства наблюдаемых автоматов. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 7–20. DOI: 10.15514/ISPRAS–2025–37(6)–1.

On Length of an Adaptive Distinguishing Sequence for a Family of Observable Finite State Machines

[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] *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, the length of an adaptive distinguishing sequence for a family of initialized complete observable, possibly nondeterministic Finite State Machines (FSM) is studied. The upper bound for the length of such a sequence is established and its reachability is shown. It is also discussed how the obtained results can be applied for constructing adaptive diagnostic test suites based on a FSM model.

Keywords: nondeterministic FSMs; adaptive distinguishing sequence; diagnostic test suites.

For citation: Burdonov I.B., Yevtushenko N.V., Kossatchev A.S. On length of an adaptive distinguishing sequence for a family of observable Finite State Machines. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1 2025, pp. 7-20 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-1.

1. Введение

В настоящей статье рассматривается задача синтеза диагностических тестов на основе модели инициального, возможно, недетерминированного конечного автомата. Предполагается, что после выполнения этапа тестирования с использованием проверяющих тестов известно конечное множество автоматов, которые соответствуют «неправильной» реализации некоторых переходов в автомате-спецификации. Для определения, какая именно реализация была предъявлена в процессе тестирования, то есть какие исправления необходимо внести в предъявленную реализацию, чтобы реализация соответствовала известной спецификации, используются диагностические тесты. Под *диагностическим* тестом понимается входная последовательность (или конечное множество входных последовательностей), по реакции на которые можно различить любые две реализации из множества конечных автоматов, зафиксированных после тестирования на соответствие спецификации, и соответственно распознать тестируемый автомат-реализацию. В работе обсуждается построение адаптивной входной последовательности, по реакции на которую можно различить любую пару из множества рассматриваемых автоматов-реализаций. В первом алгоритме для построения адаптивной различающей последовательности строится прямая сумма всех автоматов множества и исследуется возможность построения адаптивной различающей последовательности для множества начальных состояний прямой суммы. В работе оценивается наибольшая длина адаптивной различающей последовательности, если такая последовательность существует, и показывается, что если число состояний каждого из k автоматов-реализаций равно n, и адаптивная различающая последовательность для прямой суммы автоматов-реализаций существует, то длина такой последовательности может достигать величины n^k. Полученная оценка существенно меньше оценки длины различающей последовательности для произвольного автомата с nk состояниями, k из которых являются начальными [1]. Во втором алгоритме для каждой пары «реализация_1-реализация_2» строится адаптивная различающая последовательность, по поведению на которую различаются эти автоматы-реализации, и из таких различающих последовательностей «собирается» адаптивная различающая последовательность для множества всех автоматов-реализаций. Оценивается длина такой последовательности, которая не превышает величины $n^2k + k$, если число состояний каждого из k автоматов-реализаций равно n.

Структура статьи следующая. Второй раздел содержит необходимые определения и обозначения. Третий раздел описывает связь между диагностическими тестами и различающими последовательностями. В четвертом разделе устанавливаются условия существования адаптивной различающей последовательности для множества начальных состояний прямой суммы автоматов-реализаций, и оценивается длина такой последовательности (если существует). В пятом разделе предлагается алгоритм построения адаптивной различающей последовательности для множества начальных состояний автоматов-реализаций на основе «сборки» такой последовательности для пар различных реализаций. В заключении освещаются перспективы дальнейших научных исследований. В частности, отмечается, что предлагаемый подход можно использовать для синтеза диагностических тестов для определенного класса входо-выходных полуавтоматов.

2. Основные определения и обозначения

Под *конечным автоматом* (или просто *автоматом*) понимается пятерка $S = (S, I, O, h_S, S_{in})$ [2-3], где S – конечное непустое множество состояний с выделенным непустым множеством начальных состояний S_{in}, I и O – конечные непустые входной и выходной алфавиты соответственно, такие, что $I \cap O = \varnothing$, и $h_S \subseteq S \times I \times O \times S$ – *отношение* или *множество переходов*. Автомат называется *инициальным*, если S_{in} – одноэлементное множество, то есть $|S_{in}| = 1$. Автомат называется *неинициальным*, если $S_{in} = S$, и тогда в ряде случаев автомат обозначается $S = (S, I, O, h_S)$. В случае, когда $1 < |S_{in}| < |S|$, автомат называется *слабо инициальным*. Четверка $(s, i, o, s') \in h_S$ называется *переходом* в автомате в состоянии s или *переходом* из состояния s в состояние s'. Говорят, что *поведение автомата* S *определено* в состоянии s для входного символа i, если существует пара $(o, s') \in O \times S$ такая, что $(s, i, o, s') \in h_S$. Если поведение в автомате S определено в любом состоянии $s \in S$ для любого входного символа $i \in I$, то автомат называется *полностью определенным*, в противном случае автомат называется *частично определенным* или *частичным*. Автомат S называется *детерминированным*, если для любой пары $(s, i) \in S \times I$ существует не более одной пары $(o, s') \in O \times S$ такой, что $(s, i, o, s') \in h_S$. В противном случае автомат называется *недетерминированным*. Если в недетерминированном автомате S для любой тройки $(s, i, o) \in S \times I \times O$ существует не более одного состояния $s' \in S$ такого, что $(s, i, o, s') \in h_S$, то автомат называется *наблюдаемым*, в противном случае автомат называется *ненаблюдаемым*. В настоящей работе мы рассматриваем полностью определенные наблюдаемые автоматы, если явно не сказано иное.

Множество состояний $\{s' \in S \mid (s, i, o, s') \in h_S\}$ называется *io-преемником* состояния $s \in S$ для $i \in I$, $o \in O$. Вообще говоря, *io-преемник* состояния s может быть пустым множеством; в этом случае иногда говорят, что *io-преемник* состояния s *не существует*. В наблюдаемом автомате для каждого состояния $s \in S$ и каждой входо-выходной пары i/o соответствующий *io*-преемник содержит не более одного состояния. Для наблюдаемого полностью определенного автомата можно ввести функцию следующего состояния и функцию выходов: функция $next\_state_S(s, i)$ определяет возможное множество состояний автомата после подачи входного символа i в состоянии s, функция $next\_state_S^{nd}(s, i, o)$ определяет состояние автомата после подачи входного символа i в состоянии s при получении выходного символа o, и функция выходов $outs(s, i)$ определяет возможное множество выходных символов автомата после подачи входного символа i в состоянии s.

Отношение переходов обычным образом распространяется на последовательности (слова) в алфавитах I и O. Обозначим через I^* множество всех последовательностей конечной длины в алфавите I, включая пустую последовательность ε. Соответственно, функции переходов $next\_state_S$, $next\_state_S^{nd}$ и выходов $outs$ можно распространить на последовательности входных и выходных символов. В этом случае множество $next\_state_S(s, \alpha)$, $\alpha \in I^*$, включает

те и только те состояния, которые достижимы в автомате S из состояния s по входной последовательности α, то есть $next\_states(s, \alpha)$ есть α-преемник состояния s. Соответственно, множество $outs(s, \alpha)$ включает все возможные выходные последовательности (реакции) автомата S в состоянии s на входную последовательность α. Пара α/β, $\alpha \in I^*$, $\beta \in outs(s, \alpha)$, называется *входо-выходной последовательностью* автомата S в состоянии s. По определению, выходной реакцией автомата на пустую входную последовательность в любом состоянии является пустая выходная последовательность. Функция $next\_state_S^{nd}(s, \alpha, \beta)$, $\alpha = i_1 \ldots i_k$ и $\beta = o_1 \ldots o_k$, определяет $\alpha\beta$-преемник, то есть состояние, которое достижимо в автомате S из состояния s по входной последовательности α с выходной последовательностью β. Если $\beta \notin outs(s, \alpha)$, то $next\_state_S^{nd}(s, \alpha, \beta) = \varnothing$ или в этом случае говорят, что $\alpha\beta$-преемник состояния s не существует.

Состояния s_1 и s_2 полностью определенного автомата $S = (S, I, O, h_S, S_{in})$ называются *неразделимыми* или в данной работе *неразличимыми* (обозначение: $s_1 \sim s_2$) [4-5], если $\forall \alpha \in I^*$ $(outs(s_1, \alpha) \cap outs(s_2, \alpha) \neq \varnothing)$; в противном случае, состояния s_1 и s_2 называются *различимыми* (обозначение: $s_1 \nsim s_2$). Последовательность α, для которой $outs(s_1, \alpha) \cap outs(s_2, \alpha) = \varnothing$, называется *разделяющей* или *различающей* последовательностью для этих состояний. Если $|S_{in}| > 1$, то входная последовательность, которая различает каждую пару начальных состояний автомата, называется *различающей* последовательностью автомата. Автоматы $S = (S, I, O, h_S, S_{in})$ и $P = (P, I, O, h_P, P_{in})$ называются *неразличимыми* (обозначение: $S \sim P$), если для любой входной последовательности α существуют состояния $s \in S_{in}$ и $p \in P_{in}$ такие, что $outs(s, \alpha) \cap outP(p, \alpha) \neq \varnothing$; в противном случае, автоматы S и P называются *различимыми* (обозначение: $S \nsim P$). Входная последовательность α, для которой $outs(s, \alpha) \cap outP(p, \alpha) = \varnothing$ для любых $s \in S_{in}$ и $p \in P_{in}$, называется *различающей* последовательностью для этих автоматов. Заметим, что введенное определение различающей последовательности для детерминированных автоматов совпадает с хорошо известным [7-8]; для недетерминированных автоматов введенное выше понятие различимости не требует использования «всех погодных условий», то есть при анализе различимости двух недетерминированных автоматов достаточно подать различающую последовательность только один раз.

Прямой суммой полностью определенных инициальных автоматов S и P, $S = (S, I, O, h_S, s_0,)$, $P = (P, I, O, h_P, p_0)$, $S \cap P = \varnothing$, называется автомат $S \oplus P = (S \cup P, I, O, h_S \cup h_P, \{s_0, p_0\})$, и это определение естественным образом расширяется на прямую сумму k автоматов, $k > 2$. Прямая сумма инициальных автоматов $A_1 \oplus \ldots \oplus A_k$ с попарно непересекающимися множествами состояний $S_j = \{a_j^1, \ldots, a_1^{n_j}\}$, $j = 1, \ldots, k$, $k > 1$, и начальными состояниями $a_1^1, \ldots, a_k^1$, получается «склеиванием» автоматов A_j, имеет множество состояний, которое является объединением множеств S_j, $j = 1, \ldots, k$, множество начальных состояний $\{a_1{}^1, \ldots, a_k{}^1\}$, входной алфавит I и выходной алфавит O.

Как было отмечено во введении, при подаче входной последовательности, различающей два (или несколько) автоматов-реализаций, можно распознать, на какой из этих автоматов была подана эта последовательность без использования предположения о «всех погодных условиях», то есть с использованием таких различающих последовательностей можно говорить о диагностических тестах, если при тестировании на основе «белого ящика» обнаружено несоответствие тестируемой реализации спецификации. Поскольку известно, что безусловные различающие последовательности для недетерминированных автоматов могут оказаться достаточно длинными [5], то при синтезе диагностических тестов для таких автоматов имеет смысл вместо безусловных различающих последовательностей рассматривать адаптивные различающие последовательности, которые существуют чаще и обычно оказываются более короткими. Для представления адаптивных входных последовательностей используются тестовые примеры [1, 6].

Тестовый пример определяется для заданных входного и выходного алфавитов I и O. Далее инициальный автомат называется *связным*, если любое состояние автомата достижимо из начального состояния по некоторой входной последовательности. Состояние автомата называется *нетупиковым*, если в состоянии есть переход по некоторому входному символу; иначе, состояние называется *тупиковым*. *Тестовый пример* $TC(I, O)$ есть связный наблюдаемый инициальный автомат над алфавитами I и O, граф переходов которого ациклический, и в каждом нетупиковом состоянии определены переходы только по одному входному символу со всеми возможными выходными символами (в англоязычной литературе – *single-input output-complete*). По определению, если $|I| > 1$, то тестовый пример является частичным автоматом. Тестовый пример представляет адаптивную входную последовательность, и состояния s_1 и s_2 полностью определенного наблюдаемого автомата $S = (S, I, O, h_S)$ называются *адаптивно различимыми* (далее просто *различимыми*), если существует тестовый пример $TC(I, O)$, такой что каждая входо-выходная последовательность, ведущая в тупиковое состояние этого автомата, реализуется в только в одном из состояний s_1 и s_2, то есть соответствующая адаптивная входная последовательность различает состояния s_1 и s_2. Инициальные полностью определенные наблюдаемые автоматы (S, I, O, h_S, s_0) и (T, I, O, h_T, t_0) называются *адаптивно различимыми* (далее просто *различимыми*), если существует тестовый пример $TC(I, O)$, такой что каждая входо-выходная последовательность, ведущая в тупиковое состояние этого автомата, не реализуется в обоих состояниях s_0 и t_0, то есть соответствующая адаптивная входная последовательность различает автоматы S и T. Если тестовый пример является различающим для автоматов S и T, то в дальнейшем тупиковые состояния тестового примера будут помечаться символами $\bar{S}$, $\bar{T}$, $(\bar{S}, \bar{T})$. Пометка тупикового состояния символом $\bar{S}$ означает, что соответствующая входо-выходная последовательность автомата $TC(I, O)$ не реализуется в начальном состоянии автомата S, то есть такую последовательность нельзя наблюдать при подаче на автомат S адаптивной входной последовательности, задаваемой этим тестовым примером. Аналогично, пометка тупикового состояния символом $\bar{T}$ означает, что соответствующая входо-выходная последовательность автомата $TC(I, O)$ не реализуется в начальном состоянии автомата T, то есть такую последовательность нельзя наблюдать при подаче на автомат T адаптивной входной последовательности, задаваемой тестовым примером $TC(I, O)$. Согласно пометке тупикового состояния символом $(\bar{S}, \bar{T})$, соответствующая входо-выходная последовательность автомата $TC(I, O)$ не реализуется ни в начальном состоянии автомата S, ни в начальном состоянии автомата T, то есть такую последовательность нельзя наблюдать при подаче на автомат S или автомат T адаптивной входной последовательности, задаваемой тестовым примером $TC(I, O)$.

Тестовый пример $TC(I, O)$ называется *различающим* для слабо-инициального автомата $S = (S, I, O, h_S, S_{in})$, если каждая входо-выходная последовательность, ведущая в тупиковое состояние автомата $TC(I, O)$, реализуется не более, чем в одном состоянии множества S_{in}, то есть соответствующая адаптивная входная последовательность различает каждую пару начальных состояний автомата S. Для автомата S существует адаптивная различающая последовательность, если и только если для автомата S существует различающий тестовый пример [6]. Под высотой тестового примера или длиной соответствующей адаптивной входной последовательности понимается длина самой длинной последовательности из начального в тупиковое состояние, то есть длина самой длинной входной последовательности, которая будет подана на автомат в процессе различающего эксперимента. Различающие последовательности автомата-спецификации активно используются при оптимизации тестов с гарантированной полнотой на основе автоматной модели [см., например, 9 и 10], и как известно [11-12], адаптивные различающие последовательности короче безусловных и существуют чаще.

Один из подходов к синтезу различающих тестовых примеров для наблюдаемых недетерминированных автоматов базируется на понятии *адаптивной различимости* или

просто *различимости* для подмножеств состояний автомата S, которое определяется итеративно [1]. Все синглетоны множества S полагаются 0-*различимыми*. Пусть $l \geq 1$ и определены все максимальные (в смысле вложенности) $(l-1)$-различимые подмножества множества S. Подмножество S' состояний автомата S *l-различимо*, если S' есть $(l-1)$-различимое множество или существует входной символ $i \in I$, такой, что для любого $o \in O$, io-преемник множества S' пуст или является $(l-1)$-различимым множеством, причем в последнем случае любые два различных состояния из S' не могут обладать одним и тем же io-преемником. Входной символ $i \in I$, такой, что для любого $o \in O$, io-преемник множества S' пуст или является $(l-1)$-различимым множеством, причем любые два различных состояния из S' не обладают одним и тем же io-преемником, называется *различающим входным символом* для S'. Множество S' называется *адаптивно различимым* или в данной работе просто *различимым*, если S' есть l-различимое множество для некоторого натурального l. По определению различающего входного символа имеет место следующее утверждение.

Утверждение 1. Если для множества $S' \subseteq S$ существуют входной символ i и выходной символ o, такие что io-преемник множества S' есть S', то i не является различающим входным символом для S'.

Известно, что слабо инициальный полностью определенный наблюдаемый автомат S обладает различающим тестовым примером [1], если и только если множество его начальных состояний адаптивно различимо.

3. *Диагностические тесты и различающие последовательности*

При синтезе проверяющих и диагностических тестов на основе модели конечного автомата предполагается, что есть, возможно, недетерминированный автомат-спецификация S и некоторое множество его реализаций. Если в процессе тестирования выясняется, что предъявленный автомат-реализация не соответствует автомату-спецификации, причиной чего достаточно часто является «неправильная» реализация некоторых переходов в автомате-спецификации, то возникает задача обнаружения таких переходов с целью «исправления» предъявленного автомата-реализации. В этом случае необходимо решить задачу построения входной (адаптивной) последовательности, различающей каждую пару автоматов-реализаций, если такая последовательность существует, то есть построить *диагностический тест*. Предполагая, что при реализации автомата-спецификации «подозрительных» переходов, которые могут быть неправильно реализованы, не слишком много, для построения такой входной последовательности решается следующая задача. Пусть есть k реализаций автомата-спецификации, каждая из которых имеет n состояний. Если любая пара «реализация_1-реализация_2» (адаптивно) различима, то подав соответствующие входные последовательности на тестируемый автомат-реализацию, можно определить, что нужно «поправить» в соответствующем автомате-реализации. Если не все пары реализаций адаптивно различимы, то вывод можно сделать с некоторой вероятностью.

4. *Условия существования адаптивной различающей последовательности для семейства инициальных автоматов-реализаций и оценка ее длины, если такая последовательность существует*

В настоящем разделе нас интересует вопрос о существовании входной последовательности, различающей любую пару автоматов из известного семейства автоматов-реализаций. Мы далее полагаем, что семейство автоматов-реализаций, предъявленное для построения диагностического теста, содержит k инициальных полностью определенных наблюдаемых автоматов, каждый из которых, так же как спецификация, имеет n состояний. Автомат-реализация есть автомат A_j с множеством состояний $S_j = \{a_j^1, \ldots, a_j^n\}$, $n \geq 2$, $j = 1, \ldots, k$, с

начальным состоянием a_j^1. Входной и выходной алфавиты каждого автомата A_j, $j = 1, \ldots, k$, совпадают с таковыми алфавитами I и O автомата-спецификации. Без ограничения общности будем полагать, что множества состояний двух автоматов A_j, $j = 1, \ldots, k$, для различных значений j не пересекаются. Для распознавания предъявленной реализации с целью исправления в ней ошибок мы используем (адаптивную) различающую последовательность, то есть входную последовательность α, которая различает любую пару автоматов из множества A_j, $j = 1, \ldots, k$, (если такая последовательность существует). После подачи адаптивной различающей последовательности на предъявленную реализацию, по реакции на эту последовательность можно обнаружить, какой именно автомат-реализация A_j, $j = 1, \ldots, k$, требует корректировки. По определению различающей последовательности, справедливо следующее утверждение.

Утверждение 2. Входная последовательность α, которая различает любую пару автоматов из множества A_j, $j = 1, \ldots, k$, существует, если и только если существует различающая последовательность для прямой суммы автоматов A_j, $j = 1, \ldots, k$.

К сожалению, известно [5], что длина безусловной входной последовательности, различающей инициальные автоматы с n состояниями, может достигать 2^{n^2-1}, и соответственно длина различающей последовательности для прямой суммы $k \geq 2$ инициальных автоматов с n состояниями может также достигать 2^{n^2-1}, в то время как длина адаптивной различающей последовательности для двух инициальных автоматов с n состояниями равна n^2 [6, 12].

Рассмотрим прямую сумму $A_1 \oplus \ldots \oplus A_k$ из k инициальных автоматов, $k \geq 2$, в которой каждый автомат A_j, $j = 1, \ldots, k$, имеет множество состояний $S_j = \{a_j^1, \ldots, a_j^n\}$ с начальным состоянием a_j^1. Множество состояний $\{a_{j_1}^{r_1}, \ldots, a_{j_1}^{r_1}\} \in S_{j_1} \times \ldots S_{j_l}$, $1 < l \leq k$, где $j_1, \ldots, j_l \in \{1, \ldots, k\}$, попарно различны, является 1-*различимым*, если существует входной символ i, на который выходные реакции прямой суммы $A_{j_1} \oplus \ldots \oplus A_{j_l}$ различаются в любых двух состояниях множества $\{a_{j_1}^{r_1}, \ldots, a_{j_1}^{r_1}\}$. Пусть $l > 1$ и определены все $(l-1)$-различимые подмножества множества S. Подмножество $S' \in S_{j_1} \times \ldots S_{j_l}$, $1 < l \leq k$, где $j_1, \ldots, j_l \in \{1, \ldots, k\}$ попарно различны, l-*различимо*, если S' есть $(l-1)$-различимое множество или для S' существует *различающий* входной символ $i \in I$, такой, что для любого $o \in O$, io-преемник множества S' является синглетоном или $(l-1)$-различимым множеством. Если множество $\{a_1^1, \ldots, a_k^1\}$ есть l-различимое множество для некоторого натурального l, то начальные состояния $a_1^1, \ldots, a_k^1$ прямой суммы $A_1 \oplus \ldots \oplus A_k$ различимы посредством адаптивного эксперимента высоты l. Таким образом, длина адаптивной различающей последовательности для множества начальных состояний такой прямой суммы не превосходит самой длинной последовательности по io-преемникам из множества начальных состояний. Поскольку из множества начальных состояний прямой суммы достижимы только элементы декартова произведения множеств состояний автоматов-компонентов, то имеет место следующее утверждение.

Утверждение 3. Длина адаптивной различающей последовательности для прямой суммы k инициальных полностью определенных наблюдаемых автоматов с n состояниями, $n \geq 2$, $k \geq 2$, не превосходит величины n^k.

Заметим, что оценка в утверждении 3 существенно меньше оценки достижимой длины ($2^{nk} - nk - 1$) адаптивной различающей последовательности для произвольного автомата с nk состояниями, k из которых являются начальными [1]. Например, при $k = 3$, $n = 2$, имеет место $2^{nk} - nk - 1 = 57$, в то время как сумма чисел из утверждения 3 равна 2^3.

Доказательство достижимости оценки из утверждения 3 похоже на доказательство достижимости оценки для произвольного случая, и далее мы показываем, что для любого $k \geq 2$ существуют k инициальных автоматов, каждый из которых имеет n состояний, $n \geq 2$, такие,

что для прямой суммы этих автоматов кратчайший различающий тестовый пример имеет высоту n^k.[1]

Модифицируя результаты работы [1], на множестве всех подмножеств из k состояний, все состояния которых принадлежат различным автоматам A_j, вводится отношение линейного порядка, представленное цепью $C(n, k)$. Цепь $C(n, k)$ строится итеративно, начиная с $k = 1$, и по определению, $C(n, 1) = \{a_1^1\}, \{a_1^2\}, \ldots, \{a_1^n\}$. Если уже построена цепь $C(n, k)$, то для построения цепи $C(n, k+1)$ в каждое подмножество цепи $C(n, k)$ добавляется состояние a_{k+1}^1, после чего получается цепь $C^1(n, k+1)$. Если уже построена цепь $C^j(n, k+1)$, $1 < j < n$, то для получения цепи $C^{j+1}(n, k+1)$ подмножества в цепи $C^j(n, k+1)$ записываются в обратном порядке и в каждое подмножество вместо состояния a_{k+1}^j добавляется состояние a_{k+1}^{j+1}. Цепь $C(n, k+1)$ получается как конкатенация цепей $C^j(n, k+1)$, $j = 1, \ldots, k+1$.

В качестве примера рассмотрим автоматы A_1, A_2, A_3 с множествами состояний $S_1 = \{1, 2\}$, $S_2 = \{a, b\}$, $S_3 = \{A, B\}$, т.е. $n = 2$, $k = 3$. По описанным выше правилам $C(2, 1) = \{1\}, \{2\}$. Для построения цепи $C^1(2, 2)$ в каждое множество цепи $C(2, 1)$ добавляем состояние a: $C^1(2, 2) = \{1, a\}, \{2, a\}$. Чтобы построить цепь $C^2(2, 2)$ меняем порядок в цепи $C^1(2, 2)$ и вместо состояния a записываем состояние b: $C^2(2, 2) = \{2, b\}, \{1, b\}$. После конкатенации цепей $C^1(2, 2)$, $C^2(2, 2)$ получаем цепь $C(2, 2) = \{1, a\}, \{2, a\}, \{2, b\}, \{1, b\}$, в которой любые два соседних блока отличаются только одним состоянием. Аналогично, при построении $C(2, 3)$ строим цепи $C^1(2, 3)$, $C^2(2, 3)$, $C^3(2, 3)$ и их конкатенацию: $C^1(2, 3) = \{1, a, A\}, \{2, a, A\}, \{2, b, A\}, \{1, b, A\}$, $C^2(2, 3) = \{1, b, B\}, \{2, b, B\}, \{2, a, B\}, \{1, a, B\}$, и после конкатенации $C^1(2, 3)$ и $C^2(2, 3)$ получаем цепь $C(2, 3) = \{1, a, A\}, \{2, a, A\}, \{2, b, A\}, \{1, b, A\}, \{1, b, B\}, \{2, b, B\}, \{2, a, B\}, \{1, a, B\}$, в которой есть любая тройка состояний автоматов A_1, A_2, A_3, и любые соседние блока отличаются только одним состоянием.

Утверждение 4. Цепь $C(n, k)$, $1 \le n$, $1 \le k$, содержит каждое подмножество из k состояний, состоящее из состояний автоматов-компонентов, причем любые соседние блоки отличаются только одним состоянием.

Доказательство того, что в цепи $C(n, k)$ есть все подмножества из k состояний, содержащие состояния всех автоматов-компонентов, и любые два соседних блока отличаются только одним состоянием, достаточно просто проводится с использованием индукции по числу k автоматов-компонентов.

Введенное цепью $C(n, k)$ отношение линейного порядка на множестве подмножеств состояний автоматов- компонентов обозначим символом $\succ_{C(n,k)}$. Если подмножество p_{j+1} непосредственно следует за p_j в в цепи $C(n, k)$, то $p_j \succ_{C(n,k)} p_{j+1}$. По цепи $C(n, k)$, мощность которой равна n^k, построим автомат $S_{C(n, k)}$, который есть прямая сумма автоматов A_j, $j = 1, \ldots, k$, каждый из которых имеет n состояний. Число входных и выходных символов $S_{C(n, k)}$ совпадает с числом элементов в цепи $C(n, k) = p_1, \ldots, p_{C(n, k)}$. Для последнего подмножества используется специальный входной символ i_{dist}, и, кроме того, используются $(n^k - 1)$ входных символов, каждый из которых имеет вид $i_{p/p'}$ для соседних блоков p и p' цепи $C(n, k)$ (табл. 1 и 2). Множество выходных символов содержит выходные символы (0), (1), …, (k), а также все выходные символы виды <p>, $p \in C(n, k)$. Для каждой пары p и p' множеств в цепи $C(n, k)$, такой, что множество $p' = p\backslash\{j\} \cup \{r\}$ следует за p, то есть $p \succ_{C(n,k)} p'$, введем входной символ $i_{p/p'}$, по которому из состояния j есть переход в состояние r с выходным символом <0>. Кроме того, в каждом состоянии s множества $p\backslash\{j\}$ есть петля с выходным символом <0>. Таким образом, $i_{p/p}$<0>-преемником множества p является множество p'. В каждом состоянии

[1] Подобное утверждение можно сформулировать и для случая, когда автоматы прямой суммы имеют различное число состояний $n_1, \ldots, n_k$. В настоящей статье мы предполагаем, что в качестве элементов прямой суммы используются достаточно простые автоматы-мутанты, и число состояний каждого автомата-мутанта совпадает с числом состояний n автомата-спецификации.

$s \notin p$ по входному символу $i_{p/p'}$ есть петля с выходным символом <p"> для каждого $p" \in C(n, k) \backslash p$, $s \in p"$. Для состояния $s \in p \backslash \{j\}$ есть петля с выходным символом <p"> для каждого $p" \in C(n, k)$, $s \in p"$ и $p" \neq p$.

Табл. 1. Фрагмент таблицы переходов автомата $S_{C(2,3)}$ по входному символу $i_{1aA/2aA}$.
Table 1. A frame of the transition table of $S_{C(2,3)}$ under input $i_{1aA/2aA}$.

Состояние	1	2	*a*	*b*	*A*	*B*
Входной символ $i_{\{\ i_{1aA/2aA}}$	2/(0); <1*aB*>; <1*bA*>; <1*bB*>	<2*aA*>; <2*aB*>; <2*bA*>; <2*bB*>	*a*/(0); <1*aB*>; <2*aA*>; <2*aB*>	<1*bA*>; <1*bB*>; <2*bA*>; <2*bB*>	*B*/(0); <2*aA*>; <2*bA*>; <2*bB*>	<1*aB*>; <1*bB*>; <2*aB*>; <2*bB*>

Табл. 2. Фрагмент таблицы переходов автомата $S_{C(2,3)}$ по входному символу i_{dist}.
Table 2. A frame of the transition table of $S_{C(2,3)}$ under input i_{dist}.

Состояние	1	2	*a*	*b*	*A*	*B*
Входной символ i_{dist}	1/(1); <1*aA*>; <1*bA*>; <1*bB*>	2/<2*aB*>; <2*aA*>; <2*bA*>; <2*bB*>	*a*/(*a*); <1*aB*>; <1*aA*>; <2*bB*>	*b*/<1*bA*>; <1*bB*>; <2*bA*>; <2*bB*>	*A*/<1*aA*>; <2*aB*>; <2*bA*>; <2*bB*>	*B*/(*B*); <2*aB*>; <1*bB*>; <2*bB*>

Соответственно, $i_{p/p'}$<p">-преемником множества $p"$ цепи $C(n, k)$, отличного от p, является множество $p"$. Поэтому если множество p' является t-разделимым, то множество p является $(t+1)$-разделимым, поскольку по входному символу $i_{p/p'}$ из p достижимо p' с выходным символом (0), а по любому другому входному символу из p достижимо p с выходным символом <p>. Чтобы разделить состояния последнего множества $p_{C(n,k)}$ цепи $C(n, k)$ добавляется входной символ i_{dist}, который разделяет состояния последнего подмножества выходными символами (1), …, (k). Для состояния $s \notin p_{C(n,k)}$ есть петля с выходным символом <p"> для каждого $p" \in C(n, k)$, $s \in p"$. Для состояния $s \in p_{C(n,k)}$ есть петля с выходным символом <p"> для каждого $p" \in C(n, k)$, $s \in p"$ и $p" \neq p_{C(n,k)}$. Обозначим построенный таким образом автомат $S_{C(n,k)}$. По построению, для автомата $S_{C(n,k)}$ высота кратчайшего тестового примера совпадает с длиной цепи $C(n, k)$, то есть справедливо следующее утверждение.

Утверждение 5. Множество начальных состояний автомата $S_{C(n,k)}$, который имеет kn состояний, n^k входных символов и $(n^k + k + 1)$ выходных символов[2], является n^k-различимым.

Этот факт достаточно просто доказывается, поскольку входной символ $i_{p/p'}$ будет различающим входным символом только для подмножества p. Для всех остальных подмножеств $p" \neq p$ входной символ $i_{p/p'}$ не является различающим, поскольку по входному символу $i_{p/p'}$ достижимо подмножество $p"$ за счет петель по выходному символу <p"> (утверждение 1). Для последнего множества p_{last} цепи $C(n, k)$ используется входной символ i_{dist}, который среди множеств цепи $C(n, k)$ различает только состояния множества p_{last}, то есть это единственное множество цепи, которое является 1-различимым.

Утверждение 6. Начальные состояния автомата $S_{C(n,k)}$ адаптивно различимы только тестовым примером длины n^k.

В качестве примера рассмотрим автоматы A_1, A_2, A_3 с множествами состояний $S_1 = \{1, 2\}$, $S_2 = \{a, b\}$, $S_3 = \{A, B\}$, прямая сумма которых имеет множество состояний {1, 2, *a*, *b*, *A*, *B*}. Рассмотрим цепь $C(2, 3)$ = {1, *a*, *A*}, {2, *a*, *A*}, {2, *b*, *A*}, {1, *b*, *A*}, {1, *b*, *B*}, {2, *b*, *B*}, {2, *a*, *B*}, {1, *a*, *B*} и переход в автомате $S_{C(2,3)}$ под действием входного символа $i_{1aA/2aA}$. По определению, для этого входного символа будут использованы выходные символы (0), и <p">, $p" \in C(2, 3)$, $p" \neq p$.

[2] Подобно [1], число выходных символов можно сделать полиномиальным относительно числа состояний, но в этом случае доказательство утверждения 5 будет менее «прозрачным».

Непосредственной проверкой можно убедиться, что $i_{1aA/2aA}(0)$-преемником множества {1, *a*, *A*} является множество {2, *a*, *A*}. Любое другого подмножество *N* является $i_{1aA/2aA}$<*N*>-преемником самого себя. Например, множество {2, *a*, *A*} есть $i_{1aA/2aA}$<2*aA*>-преемник самого себя. Для входного символа i_{dist} фрагмент таблицы переходов автомата $S_{C(2,\ 3)}$ имеет вид таблицы 2. Множество {2, *a*, *B*} является 1-различимым входным символом i_{dist}; любое другого подмножество *N* является i_{1dist}<*N*>-преемником самого себя.

Согласно утверждению 6, при построении прямой суммы всех *k* автоматов A_j, высота различающего тестового примера, то есть длина адаптивной различающей последовательности, для множества начальных состояний прямой суммы может достигать n^k, то есть экспоненциальная оценка длины адаптивной различающей последовательности для множества начальных состояний автоматов-компонентов относительно числа состояний автоматов-компонентов является достижимой. Поэтому ниже мы предлагаем рассмотреть несколько другой подход к построению адаптивной различающей последовательности. Подход основан на «надстраивании» определенным образом различающих последовательностей для пар автоматов-мутантов, для которых высота кратчайшей адаптивной различающей последовательности не превосходит n^2.

5. *Построение адаптивной различающей последовательности для семейства инициальных автоматов-реализаций на основе различающих последовательностей для пар автоматов-реализаций*

Если автоматы-компоненты A_j, $j = 1, \ldots, k$, попарно различимы, то для каждой пары инициальных автоматов $A_1, \ldots, A_k$, $k \geq 2$, существует различающий тестовый пример. Для пары A_j, A_r, $j, r = 1, \ldots, k$, $j \neq r$, рассмотрим кратчайший различающий тестовый пример $TC(A_j, A_r)$, тупиковые вершины которого помечены $\bar{A}_j$, $\bar{A}_r$, ($\bar{A}_j$, $\bar{A}_r$). Пометка тупикового состояния $\bar{A}_j$ ($\bar{A}_r$, ($\bar{A}_j$, $\bar{A}_r$)) в тестовом примере $TC(A_j, A_r)$ означает, что наблюдаемая входо-выходная последовательность из начального состояния в данное тупиковое состояние не принадлежит автомату A_r (автомату A_j, автоматам (A_j, A_r). Поскольку каждый из тестовых примеров является ациклическим автоматом, все тестовые примеры можно собрать в единое тестовое множество (или тест), добавляя в каждом тупиковом состоянии множества, построенного для уже рассмотренных тестовых примеров, сначала символ «сброс», а затем начальное состояние следующего тестового примера. В этом случае окончательный тестовый пример будет «собран» из $k(k - 1)/2$ тестовых примеров для пар автоматов-реализаций, высота каждого из которых не превосходит n^2. В данном разделе мы рассматриваем этот подход, когда на автомат-реализацию адаптивно подаются тестовые примеры $TC(A_j, A_r)$, разделенные сигналом «сброс».

Таким образом, предлагается рассматривать множество $TC = \{TC(A_j, A_r): j, r = 1, \ldots, k, j \neq r\}$. Множество $TC = \{TC(A_j, A_r): j \neq r\}$ содержит тестовый пример для любой пары A_j, A_r, $j \neq r$, поэтому после подачи всех тестовых примеров из множества автоматов A_j, $j = 1, \ldots, k$, останется только один автомат A_m, который не появился с пометкой $\bar{A}_m$ после подачи любого тестового примера. Этот автомат A_m и описывает предъявленную реализацию.

Adaptive_identification { /* **Адаптивная идентификация предъявленного автомата-реализации *A* из семейства** из множества {$A_1, \ldots, A_k$}, $k \geq 2$, **инициальных полностью определенных наблюдаемых автоматов посредством подачи входных последовательностей**
*/

| ***Для каждой пары автоматов*** $A_j, A_r \in \{A_1, \ldots, A_k\}$, $j, r = 1, \ldots, k, j \neq r$,

| ***Построить*** кратчайший различающий тестовый пример $TC(A_j, A_r)$, тупиковые вершины которого помечены $\bar{A}_j$, $\bar{A}_r$, ($\bar{A}_j$, $\bar{A}_r$);

Если для некоторой пары автоматов различающего тестового примера не существует
выдать сообщение «предъявленный автомат A нельзя идентифицировать однозначно»;
конец алгоритма;
Иначе
M: = $\{A_1, \dots, A_k\}$;
Пока $|M| > 1$
Выбрать пару автоматов A_j, $A_r \in M$, $j \neq r$;
Подать тестовый пример $TC(A_j, A_r)$ на тестируемый автомат A;
Если в $TC(A_j, A_r)$ достижимо состояние, помеченное $\bar{A}_j$,
M: = $M \setminus A_j$;
Если в $TC(A_j, A_r)$ достижимо состояние, помеченное $\bar{A}_r$,
M: = $M \setminus A_r$;
Если в $TC(A_j, A_r)$ достижимо состояние, помеченное $(\bar{A}_j, \bar{A}_r)$,
M: = $M \setminus \{A_j, A_r\}$;
Конец цикла «Пока»;
Если $M = \{A_m\}$
A эквивалентен A_m /* для идентификации предъявлен автомат A_m */
Конец алгоритма; }

Пример. Пусть $M = \{A_1, A_2, A_3, A_4\}$, для каждой пары различных автоматов из M существует различающий тестовый пример, и для распознавания предъявлен автомат A_2.
Если на распознаваемый автомат подан тестовый пример $TC(A_1, A_2)$, то наблюдаемая входо-выходная последовательность соответствует автомату A_2, т.е. достигается состояние $\bar{A}_1$.
Удаляем A_1 из множества M: $M = \{A_2, A_3, A_4\}$.
Пусть далее на распознаваемый автомат A_2 подан тестовый пример $TC(A_3, A_4)$.
Если наблюдаемая входо-выходная последовательность не соответствует ни автомату A_3, ни автомату A_4, то в тестовом примере после наблюдения входо-выходной последовательности будет достигнуто состояние $(\bar{A}_3, \bar{A}_4)$. Из множества M удаляются автоматы A_3 и A_4: $M = \{A_2\}$, то есть для распознавания предъявлен автомат A_2.
Если наблюдаемая входо-выходная последовательность соответствует автомату A_3, то, поскольку подается тестовый пример $TC(A_3, A_4)$, входо-выходная последовательность не соответствует автомату A_4. Поэтому в тестовом примере после наблюдения входо-выходной последовательности будет достигнуто состояние $\bar{A}_4$: $M = \{A_2, A_3\}$, и т.д.
Утверждение 7. Если в результате выполнения алгоритма не выдается сообщение «предъявленный автомат A нельзя идентифицировать однозначно», то автомат A идентифицируется однозначно, и длина тестовой последовательности для идентификации автомата A не превышает $n^2k + k$.

Доказательство. Автомат A идентифицируется однозначно, поскольку наблюдаемые входо-выходные последовательности не соответствуют реализациям, удаляемым из множества M. Оценка высоты кратчайшего тестового примера, равна n^2, количество используемых тестовых примеров не превосходит $(k - 1)$, между которыми подается сигнал «сброс».
Согласно оценке длины идентификационной последовательности (утверждение 7), второй подход к идентификации автомата, описывающего поведение предъявленной реализации, в общем случае оказывается более эффективным. Тем не менее, если существует, например, входной символ, различающий все пары начальных состояний автоматов-реализаций, то именно он и будет искомым тестовым примером. Поэтому обсуждение свойств множества автоматов-реализаций, для которых более удобным является каждый из подходов, требует дальнейших исследований. Кроме того, построение одного различающего тестового примера представляет интерес для решения других задач идентификации состояний.

6. *Заключение*

В настоящей статье авторы обсудили возможность построения диагностических тестов для семейства автоматов, которые являются наиболее «подозрительными» для наличия ошибок после проверки тестируемой реализации на соответствие спецификации. Показано, что длина различающей адаптивной последовательности для множества из k таких автоматов с n состояниями, $k > 1$, $n > 1$, может достигать величины n^k, как известно, для безусловных различающих последовательностей достижимая оценка еще выше. Для понижения достижимой оценки в работе предлагается еще один алгоритм построения адаптивной различающей последовательности, основанный на «надстраивании» определенным образом различающих последовательностей для пар автоматов-мутантов и повторном использовании сигнала «сброс».

Если для множества «подозрительных» автоматов-реализаций различающая последовательность не существует, то есть если в множестве существуют две адаптивно неразличимые реализации, то имеет смысл говорить об установлении допустимого множества автоматов, которому может принадлежать предъявленный автомат-реализация. Автоматы из этого множества можно различить более короткими тестами относительно эквивалентности и/или редукции, используя предположение «о всех погодных условиях». Альтернативой этому может быть идентификация предъявленного автомата-реализации с некоторой вероятностью. В последующих работах авторы предполагают исследовать оба направления.

Авторы также отмечают, что результаты работы могут быть использованы для синтеза диагностических тестов для входо-выходных полуавтоматов [13], что является еще одним направлением будущих исследований.

Список литературы / References

[1]. Khaled El-Fakih, Nina Yevtushenko, Natalia Kushik. Adaptive distinguishing test cases of nondeterministic finite state machines: test case derivation and length estimation. Formal Aspects Comput., 2018, 30(2), pp. 319-332.

[2]. Tiziano Villa, Timothy Kam, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Synthesis of Finite State Machines. Springer, 1997.

[3]. Villa T, Yevtushenko N, Brayton R, Mishchenko A, Petrenko A, Sangiovanni-Vincentelli A. The Unknown Component Problem: Theory and Applications, 2012. Springer.

[4]. P. Starke, Abstract Automata. American Elsevier, 1972.

[5]. Spitsyna N. Studying the separability relation between finite state machines. Softw. Test., Verif. Reliab., 2007, (17(4), pp. 227-241.

[6]. Alexandre Petrenko, Nina Yevtushenko. Conformance Tests as Checking Experiments for Partial Nondeterministic FSM. International Workshop on Formal Approaches to Software Testing (FATES), 2005, pp. 118-133.

[7]. Moore E. F. Gedanken-experiments on sequential machines. In Automata Studies (Annals of Mathematical Studies), 1956, 1, pp. 129-153.

[8]. Гилл А. Введение в теорию конечных автоматов. Наука, 1966.

[9]. Chow T. S. Testing software Design Modelled by Finite State Machines. IEEE Trans. Software Eng., 1978, 4(3), pp. 178-187.

[10]. D. Lee, M. Yannakakis. Testing Finite-State Machines: State Identification and Verification. IEEE Transactions on Computers, 1994, 43(3), pp. 306-320.

[11]. R. Alur, C. Courcoubetis, M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines. Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, 1995, pp. 363-372.

[12]. H.Yenigun, N. Yevtushenko, N. Kushik, J. López. The effect of partiality and adaptivity on the complexity of FSM state identification problems. Труды ИСП РАН, 2018, 30(1), с. 7-24.

[13]. N.Yevtushenko, I.Burdonov, A.Kossachev. Deriving Distinguishing Sequences for Input/Output Automata. Proceedings of the 2020 IEEE East-West Design & Test Symposium, 2020, pp. 1-5.

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.

DOI: 10.15514/ISPRAS-2025-37(6)-2

Разновидность JavaBeans-компонент: композиция типов из агрегации инстансов

Е.М. Гринкруг, ORCID: 0000-0001-6740-0541 <grinkrug@ispras.ru>

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

Аннотация. Представлен подход к реализации компонент JavaBeans, который обеспечивает создание определяемых пользователем компонент без их компиляции, путем манипуляций с существующими компонентами. Компонентная модель JavaBeans содержит принципиальные ограничения. Компоненты в ней являются классами, определенными для манипулирования их инстансами в предназначенной для этого среде манипулирования. Цель манипуляций – достичь требуемых состояний инстансов компонент и поведения их агрегации в целом; готовая агрегация может быть сериализована и десериализована позже в аналогичной среде. Тут скрыто противоречие: начиная с использования набора инстанциируемых классов, мы в итоге приходим к копированию агрегации из их инстансов. Чтобы использовать определяемую пользователем агрегацию для получения нового составного компонента, требуется сгенерировать его класс, подменяющий парадигму программирования (с инстанциирования на копирование). Предложено расширение компонентной модели JavaBeans, позволяющее динамически создавать определяемые пользователем компоненты без их кодогенерации и копирования агрегации инстансов.

Ключевые слова: компонентная модель; компонент; тип; прототип; инстанс; свойства; интерфейс; реализация типа.

Для цитирования: Гринкруг Е.М. Разновидность JavaBeans-компонент: композиция типов из агрегации инстансов. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 21–42. DOI: 10.15514/ISPRAS–2025–37(6)–2.

Yet Another Kind of JavaBeans: Composed Types from Aggregated Instances

E.M. Grinkrug, ORCID: 0000-0001-6740-0541 <grinkrug@ispras.ru>

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

Abstract. The paper presents an approach to the JavaBeans-components implementations so that they provide support for dynamic components composition – user defined components creation without compiling them, by manipulations with pre-existing components instead. JavaBeans component model, presented at the beginning of Java technology, has its' limitation that seems to be significant. The JavaBean component, by definition, is a serializable Java class with public no-arguments constructor; additionally, JavaBeans design patterns serve to use JavaBeans-components instances in a specific manipulating environment. The goal of the manipulations is to achieve the required states of the instances and their aggregation behavior altogether, when the aggregation can be serialized and deserialized later in similar environment. There is a hidden contradiction: starting from a predefined set of JavaBeans components – classes for class-based object-oriented environment, we end up with a prototype-based style for instances aggregation usage. To use user-defined aggregation as a new composed component we must change a programming paradigm and generate code in static-like way of component creation. We propose an evolution of JavaBeans component model that enables user defined composed components creation dynamically without aggregation cloning.

Keywords: component model; component; type; prototype; instance; properties; interface; type implementation.

For citation: Grinkrug E.M. Yet Another Kind of JavaBeans: Composed Types from Aggregated Instances. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 21-42 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-2.

1. Введение

Компонентно-Ориентированное Программирование (КОП), как понятие, стало официально использоваться с 1968 года, после конференции [1], где обсуждались основные отличия разработки компьютерных программ от разработки аппаратных средств самих компьютеров. Основным был вопрос: почему аппаратура компьютера создается быстрее, чем программное обеспечение для него? Ответом было признание того, что компьютеры проектируются из готовых компонент, чего нельзя сказать о программах (и до сих пор).

С тех пор было предложено множество программных технологий и компонентных моделей для внедрения КОП в практику разработки программного обеспечения. Компонентная модель – это совокупность правил, определяющих, что является компонентом [2] и как такие компоненты используются совместно [3].

В любом компьютере программа представляет собой набор экземпляров взаимосвязанных компонент, начиная с уровня машинного языка, где программы – это наборы машинных инструкций разных типов (определенных системой команд). В этом смысле любая программа является композицией экземпляров компонент, и не-компонентного программирования не бывает. Проблема заключается в том, как именно определяются сами компоненты и композиции с их использованием. Иными словами – в том, какая компонентная модель используется.

Мы ограничим обсуждение программированием на Java-платформе, для которой была предложена компонентная модель JavaBeans [4], лежащая в основе многих современных технологий. Фирма Sun Microsystems изначально указывала в документации: «JavaBeans component model is the only component model for the Java-machine», хотя впоследствии появились и другие компонентные модели. Основные соображения могут быть полезны и для других, сходных с Java-платформой, средств программирования.

Предпосылки работы возникли из опыта реализации средствами Java-программирования подмножества языка моделирования виртуальной реальности VRML-97 [5]. Реализация средств моделирования средствами объектно-ориентированного программирования (для моделирования и задуманного) является естественной и наглядной: результат моделирования непосредственно отображается в 3D-графике.

Все базовые компоненты стандарта VRML были реализованы как стандартные JavaBeans-компоненты; другие JavaBeans-компоненты являлись элементами графического интерфейса пользователя для отображения трехмерной сцены. Это позволило выполнять моделирование и визуализацию в стандартных инструментах манипулирования такими компонентами и наблюдать достоинства и недостатки компонентной модели JavaBean.

Развитию возможностей и преодолению недостатков этой компонентной модели посвящена данная работа. Она не претендует на полноту изложения, но посвящена соображениям, на которых основана реализация расширения компонентной модели.

В разделе 2 дан обзор исходной компонентной модели, ее использования и соображения о направлении развития. В разделе 3 изложено основное содержание работы - принципы реализации базовых компонент и их использование при прототипировании составных компонент. Перечень основных положений работы и их места среди прочих подходов приведены в разделах 4 и 5; заключают работу перспективные цели проекта.

2. Компонентная модель и ее развитие

Для дальнейшего обсуждения нам понадобятся сведения об исходной компонентной модели JavaBeans и используемой терминологии.

2.1 Компонентная модель JavaBeans

2.1.1 Определение и важные особенности

Компонентная модель JavaBeans дает весьма общее определение своих компонент [4]: JavaBeans–компонент – это общедоступный (public) класс (возможно со вспомогательными классами и ресурсами), который:

- имеет общедоступный конструктор без аргументов, и
- реализует маркерный интерфейс java.io.Serializable.

Возможность создавать объекты такого класса без предоставления дополнительной информации конструктору позволяет получать «одинаковые» *инстансы* компонента «бесконтекстным» образом (без зависимости от контекста их создания) и определяет *универсальный способ инстанциирования* таких компонент.

Специализация этих инстансов обеспечивается вызовами их общедоступных методов, среди которых особо выделяются *методы доступа к значениям свойств* (*property accessors*), хотя их наличие в классе по определению JavaBeans-компонента не является обязательным. Эти методы (*setters, getters*) позволяют абстрагироваться от наличия соответствующих полей класса для хранения значений свойств вообще, а при их наличии – варьировать способы доступа к значениям, чего сами поля выразить не позволяют.

Требование реализации интерфейса java.io.Serializable исторически связано с сохранением и восстановлением инстансов компонент с помощью встроенных в JVM базовых средств сериализации/десериализации; оно не является обременительным при наличии иных средств.

Взаимодействие с инстансами JavaBeans-компонент возможно путем вызовов их (public) методов и/или с использованием реакций на события, которые вырабатываются инстансами компонент (в соответствии с событийной моделью Java-платформы). В основе реализации лежит механизм интроспекции их классов, опирающийся на средства рефлекции.

Возможной реакцией на событие является вызов метода некоторого инстанса компонента с передачей ему информации из события. Это называют связыванием события объекта-источника с вызовом метода у объекта-приемника. Для интерфейсов общего вида при этом изначально требовалась генерация кода классов объектов-посредников, которые подписывались на получение событий нужного типа и делегировали реакцию к указанному методу; с появлением механизма Dynamic Proxy (в JDK1.3) реализация упростилась (давая наглядный пример того, как упрощается использование компонент при добавлении динамических средств в Java-платформу).

Для событий, сообщающих об изменении значения свойства (*property*), где реакция на событие сводится к передаче значения свойства инстанса-источника в заданное свойство инстанса-получателя, предоставляются готовые объекты-посредники; а само свойство источника называется «связываемым» (*bound property*).

2.1.2 Использование JavaBeans-компонент

Сценарии использования JavaBeans-компонент демонстрировал Bean Development Kit (BDK), прилагавшийся к спецификации [4] с интерактивным инструментом BeanBox. Они таковы:

- в BeanBox поставляются архивы с JavaBeans-компонентами в виде байткодов их классов; BeanBox их загружает и показывает список доступных в нем компонент;
- BeanBox предоставляет инстанс (одноименного) контейнера для агрегирования инстансов загруженных компонент (классов); инстансы создаются бесконтекстным образом (default-конструкторами);
- созданные инстансы отображаются в GUI инстанса контейнера BeanBox;
- выполняется настройка значений свойств созданных инстансов; интерактивная настройка предполагает наличие редакторов значений свойств; сам BeanBox располагает лишь базовым их набором, но компоненты могут их «приносить с собой»; при наличии нужных редакторов могли бы интерактивно выполняться «внедрения зависимостей» (dependency injections) инстансов друг от друга, но сам BeanBox это не поддерживал;
- выполняется настройка графов распространения событий, которые связывают события от инстансов-источников с методами реакции на события инстансами-приемниками; такие графы определяют совместное управляемое событиями (event-driven) поведение создаваемой агрегации инстансов.

Приложение BeanBox предоставляло средства для сохранения и загрузки контента инстанса контейнера с помощью стандартной сериализации/десериализации, для чего требование реализации интерфейса java.io.Serializable и входило в определение JavaBeans-компонент. Приложение BeanBox демонстрировало также вариант программной генерации и компиляции класса контейнера, реализовывавшего предопределенный интерфейс (на примере устаревшего класса java.applet.Applet), в инстанс которого можно было загрузить (десериализовать) созданную агрегацию инстансов компонент.

Компонентная модель JavaBeans используется как в модулях JDK, так и в популярных современных библиотеках. В современных JDK появились средства для более эффективной реализации этой компонентной модели, однако, имеются важные архитектурные причины и для совершенствования её самой.

Все компоненты JavaBeans являются Java-классами, загруженными загрузчиками классов и представленными на этапе исполнения (at runtime) объектами типа java.lang.Class. Эти объекты-классы, способные создавать инстансы бесконтекстным образом, сами (по определению) бесконтекстным образом создаваться не могут (их класс java.lang.Class не

является JavaBeans-компонентом). Мы можем выполнить агрегацию инстансов компонент внутри инстанса компонента-контейнера (предопределенного типа), но мы не можем создать новый тип такого контейнера в динамике, без генерации его байткода и без помощи загрузчика класса, превращающего этот байткод в объект-класс. Мы можем лишь копировать содержимое из одного инстанса компонента-контейнера (имеющегося класса) в другой, подменяя, в сущности, инстанциирование (отсутствующих без их кодогенерации) классов *составных компонент* копированием (клонированием) содержимого инстансов имеющихся классов-контейнеров (включая копирование через сериализацию/десериализацию).

Имея изначально набор базовых компонент в виде классов, созданных на основе парадигмы ООП (*class-based object-oriented programming*), мы приходим при их использовании к клонированию объектов-прототипов. Вынужденно обходя отсутствие возможности создать составной компонент в динамике и заменяя его инстанциирование клонированием инстансов имеющихся компонент-контейнеров, *мы (неявно) подменяем парадигму программирования*.

Кроме того, поскольку JavaBeans-компонент – это *произвольный* (сериализуемый) *класс, который инстанциируется бесконтекстно*, его нельзя адаптировать под конкретный контекст использования: инстансы компонента, используемые в разных контекстах, создаются «одинаковыми»; мы при их инстанциировании не используем информацию об их применении в конкретном контексте реализации агрегата из них, а это может быть весьма полезно.

2.2 Технология использования компонент

Технологии использования готовых компонент при создании изделий из них обладают рядом характерных черт, которые полезно сопоставить с использованием программных компонент. При этом важно явно отличать типы используемых объектов от самих этих объектов – инстансов этих типов (к сожалению, смешение понятий классов и создаваемых ими объектов-инстансов встречается даже в официальной документации).

Конструкторы (как наборы заготовок для сборки изделий, а не программные средства инициализации объектов) обычно содержат некоторое количество деталей различных типов; эти детали могут соединяться друг с другом заранее определенным способом в заранее заготовленных местах. При сборке конкретного изделия не все имеющиеся возможности соединений деталей используются: остаются избыточные места для соединений, что «ухудшает» готовое изделие из-за оказавшейся лишней универсальности его деталей.

Разной может быть и технология изготовления самих деталей. При литье используется «гибкий» материал (например, пластилин), по гибкому образцу делается твердая форма (например, гипсовая), используемая для получения «монолитных» изделий.

Сопоставим эти соображения с использованием программных компонент - классов. Традиционно, они «описываются» на объектно-ориентированном языке. Их автор «держит в голове» их состав, понимая, как с помощью языка выразить описание создаваемого типа (класса) из его элементов так, чтобы в будущем (после компиляции, генерации кода и его загрузки) инстансы этого класса (экземпляры создаваемых им в динамике объектов), давали и/или делали то, что он задумал. Это поддерживают все традиционные технологии объектно-ориентированного (class-based) программирования, и в этом их сложность.

Аналогично, разработчик аппаратуры (например, радиоприемника), если он достаточно опытен, может сразу нарисовать принципиальную схему новой модели (радиоприемника), где укажет, какие именно детали и как именно должны быть собраны и соединены между собой (например, паяльником) на монтажной плате. При этом:

- сама эта принципиальная схема работать (звучать) не будет, но на заводе по этой схеме, если она правильная, обеспечат производство работоспособных изделий;

- иногда даже опытному инженеру надо попробовать разные варианты, сделав опытные экземпляры – прототипы, прежде чем остановиться на искомой конструкции: ему надо убедиться в ее качестве.

Если *работающий прототип* есть, дальше – дело техники: по нему можно нарисовать схему. Некоторые детали прототипа могут настраиваться (например, конденсаторы переменной емкости), а в схеме можно указать уже определенное на этапе настройки прототипа значение (например, оптимальное значение емкости).

Наконец, еще одно важное замечание: технология изготовления прототипа из компонент, как и технология тиражирования инстансов компонент (созданных по схеме, проверенной прототипом) отличаются от технологии изготовления самих компонент для прототипа (свинчивание готовых деталей, монтаж изделия из электронных компонент, и т.п.).

С этой точки зрения, компиляция кажется исключением: компилятор может «одинаково успешно» производить компоненты с любой сложностью их внутренней организации, но:

- он обычно делает это «в статике», а не «в динамике», и
- для работы ему необходима исходная информация от автора кода: *чтобы компилировать, надо иметь, что компилировать*.

Известны программные инструменты (часто это GUI-построители), которые в процессе конструирования прототипа сохраняют («под капотом») некоторый код его создания, который потом используется при компиляции конечного результата. При этом, однако, конструируемый прототип не обладает полной функциональностью законченного изделия и не демонстрирует непосредственно желаемого в итоге поведения; это получается на этапе исполнения только в результате последующей компиляции, загрузки и т.д.

Мы хотим иметь возможность наблюдать и настраивать *полнофункциональный* прототип, из которого автоматически извлекается тип, будущие инстансы которого повторяют функциональность, заложенную в прототипе. При такой настройке прототипа мы хотим уметь отказываться от избыточной для конкретного применения универсальности используемых компонент.

3. Принципы реализации и композиции компонент

Нас интересует компонентная технология, позволяющая в динамике производить новые типы – компоненты. Такая технология должна быть «логически замкнутой объектно-ориентированной технологией», которая поддерживает создание *составных типов* (*составных компонент*), без использования компиляции (кодогенерации) и не подменяет инстанциирование типов клонированием их инстансов.

Это пожелание требует расширения возможностей компонентной модели JavaBeans, где типы объектов (представленные классами) появляются на этапе исполнения только в результате генерации байткодов и их загрузки. Если компонент – это тип создаваемых им инстансов, надо уметь создавать новые составные типы, оставаясь в рамках исходной объектно-ориентированной парадигмы: так, чтобы эти составные типы создавали свои (составные) инстансы, а не клонировали объекты-прототипы (что, в частности, способствует повторному использованию/разделению общих составляющих объектов).

Нас интересуют принципы композиции компонент, реализуемые *имеющимися средствами* JDK (возможно, они будут расширены с внедрением проекта Babylon [6]). Мы опираемся на уже имеющиеся средства работы с JavaBeans-компонентами, и *эволюционируем* эту компонентную модель, оставаясь в ее рамках (реализуя наши компоненты как *специфические* JavaBeans-компоненты).

Основной принцип «перекликается» с известной проблемой «курица или яйцо»: что первично – объект или порождающий его тип? Мы хотим обеспечить динамическую

эволюцию типов, предполагая возможность их создания из уже имеющихся типов путем агрегирования их инстансов в инстансе специального контейнера-прототипа, из которого автоматически извлекается необходимая «генетическая информация», характеризующая *новый тип – как композицию, созданную из типов, инстансы которых агрегировались в прототипе.*

В JVM первичным является класс: все, что можно выразить на исходном языке, представляет собой описание, которое на этапе исполнения представлено классом. Мы надстраиваем систему типов в рассматриваемой компонентной модели так, чтобы типы появлялись не только в результате работы загрузчика байткодов классов, но и в результате *преобразования прототипа в тип*. Типы остаются первичными: объекты-инстансы могут создаваться только типами, и каждый объект позволяет узнать его тип (включая сами типы). В первую очередь нас интересуют *типы-компоненты – типы, порождающие свои инстансы бесконтекстным образом.*

Мы предоставляем библиотеку, расширяющую компонентную модель JavaBeans новыми JavaBeans-компонентами, поддерживающими обобщение понятия типа над классами JVM.

Будем называть эту надстройку BeanVM (реализуя ее с помощью JavaBeans-компонент). BeanVM допускает две разновидности типов объектов: типы, реализованные классами JVM (или *типы-классы*), и типы, полученные в результате превращения объекта-прототипа (как агрегата инстансов компонент внутри инстанса специального контейнера-прототипа) в тип (как композицию из типов элементов, агрегированных в инстансе прототипа).

При агрегировании инстансов компонент в инстансе контейнера прототипа можно настраивать их состав, свойства и взаимодействия. Полученный из прототипа *составной тип* (*composed type*) является законченной композицией составляющих его типов. Каждый *составляющий тип* (*composing type*) полученной композиции является типом, описывающим («уточняющим») использование некоторого исходного типа (компонента) в контексте реализации составного типа (компонента). Аналогично типам (классам) JVM, все типы BeanVM являются immutable-объектами (инстансы могут ими не быть).

Соотношение компонент JavaBeans и компонент BeanVM между собой показано на рис. 1. Компонент в BeanVM – это тип BeanVM, инстанциируемый бесконтекстно. Для JavaBeans-компонент, не являющихся BeanVM-компонентами предоставляется адаптер, позволяющий взаимодействовать с ними средствами взаимодействия с инстансами BeanVM-компонент. Показанные на рис. 1 composed-компоненты не имеют собственного класса реализации и определяются пользователем (с помощью BeanVM API).

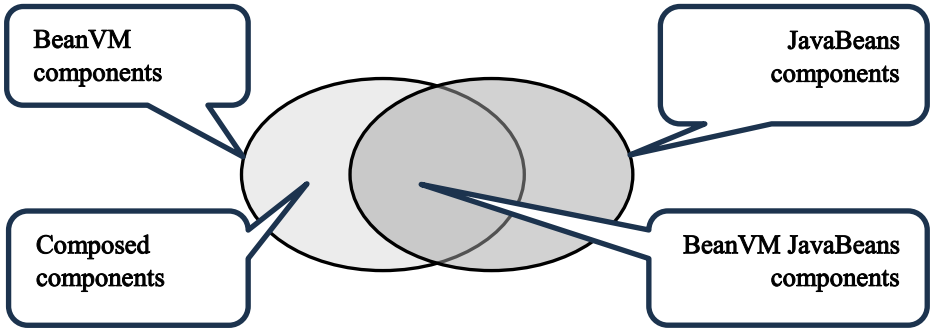


Рис. 1. Соотношение JavaBeans-компонент и BeanVM-компонент.
Fig. 1. JavaBean-components and BeanVM-components interrelation.

3.1 Типы и инстансы BeanVM

Все типы BeanVM доступны на этапе исполнения с помощью операции BeanVM API Type.forName(<имя_типа>), где имена типов отслеживаются экземплярами загрузчиков типов

BeanVM аналогично тому, как это делается в JVM при получении класса по имени в операции Class.forName(<имя_класса>). Операция Type.forName() сперва ищет одноименный java-класс с помощью загрузчика класса и операции Type.forClass(<имя_класса>). При первом выполнении этой операции для данного класса происходит создание и регистрация в загрузчике типов *типа BeanVM для данного класса* (т.е. соответствующего *типа-класса*), который и выдается в качестве результата операции; последующие поиски типа BeanVM по этому классу JVM будут выдавать уже зарегистрированный экземпляр типа-класса. Проблема расширенной типизации решается внедрением дополнительной косвенности: тип-класс BeanVM является «объектом-оберткой» java-класса его реализации.

При отсутствии класса с указанным именем (при отсутствии возможности предоставить соответствующий тип-класс) загрузчик типов BeanVM предпринимает попытку создания и регистрации составного типа из файла-ресурса с соответствующим именем, хранящего его описание. Регистрируемый при этом тип BeanVM является результатом создания инстанса прототипа с информацией, предоставленной парсером его описания из файла-ресурса, и последующего преобразования этого инстанса прототипа в составной тип.

Все внешние взаимодействия с инстансами BeanVM-компонент происходят в терминах операций с их свойствами (properties). Но эти свойства в качестве своих значений содержат объекты, имеющие тип BeanVM (property value type), а не класс JVM (property value class). Это позволяет распространить контроль типов значений свойств на все типы BeanVM.

Для внешнего взаимодействия с инстансами компонент и их свойствами предоставляются следующие операции, сходные с методами доступа к свойствам для JavaBeans-компонент (*внешний* Bean API):

```
public final void setPropertyValue(String propertyName, Object newValue);
public final void setPropertyValue(String propertyName, int elementIndex, Object newValue);
public final Object getPropertyValue(String propertyName);
public final Object getPropertyValue(String propertyName, int elementIndex);
public final void addPropertyChangeListener(String propertyName, PropertyChangeListener listener);
public final void removePropertyChangeListener(String propertyName, PropertyChangeListener listener);
// следующие операции действуют сразу для всех свойств инстанса компонента:
public final void addPropertyChangeListener(PropertyChangeListener listener);
public final void removePropertyChangeListener(PropertyChangeListener listener);
```

Здесь elementIndex соответствует индексу, который используется в методах доступа к значениям индексированных свойств (indexed properties) JavaBeans-компонент; методы регистрации/дерегистрации объектов-получателей событий об изменениях значений свойств полностью соответствуют средствам реализации связываемых свойств (bound-properties) JavaBeans. Для ускорения обращений вместо строки имени свойства (propertyName) можно использовать целочисленный индекс свойства (propertyIndex), получив его из типа заранее.

Указанные выше операции контролируются в динамике с использованием типов свойств, где содержится информация о возможностях выполнения операций доступа к ним для записи значения (write access: скалярный (W) и/или индексированный (w)), чтения значения (read access: скалярный (R) и/или индексированный (r)) и регистрации/дерегистрации объектов-получателей событий об изменениях значений свойств (bind access (B)). При операциях записи динамически контролируется тип записываемого значения. Нарушения приводят к исключительным ситуациям - разновидностям RuntimeException.

Компоненты JavaBeans являются «черными ящиками» с внешними интерфейсами, определяемыми с помощью интроспекции (рефлекции) их классов. Мы сохраняем основные средства JavaBeans-компонент, но наделяем их некоторыми характерными для BeanVM особенностями: реализации инстансов всех компонент BeanVM должны наследоваться (прямо или опосредованно) из предоставленного базового класса Bean (здесь мы используем короткие имена типов). Это позволяет рассматривать инстансы таких компонент как «серые ящики», используя внутреннюю реализацию их свойств суперклассом Bean.

Инстансы компонент BeanVM с реализациями, унаследованными из базового класса JVM Bean, обладают «самореализуемыми» свойствами (по описаниям их типов). При этом инстансы компонент-классов снабжены присущими им поведениями (реакциями на изменения значений свойств, определенными в их классах реализации); поведение инстансов составных компонент определяется и реализуется совместными поведениями инстансов компонент их внутренней реализации.

3.2 Компоненты-классы в BeanVM

При первом «знакомстве» BeanVM с классом создается и регистрируется в загрузчике типов соответствующий классу тип (тип-класс). Такое «первое знакомство» может произойти при явном вызове метода получения типа по классу, либо при первом инстанциировании класса реализации компонента, либо при использовании класса в качестве JVM-типа значения какого-либо свойства какого-либо класса, для которого определяется тип.

Имя типа-класса заимствуется из его класса реализации. Если класс является реализацией компонента BeanVM, предоставляется описание его интерфейса в терминах типов его свойств (propertyTypes). Класс реализации компонента подвергается интроспекции, которая предоставляет стандартный массив дескрипторов свойств (PropertyDescriptor'ов и/или IndexedPropertyDescriptor'ов); из них создается массив объектов типа PropertyType, каждый из которых содержит имя свойства (propertyName), тип значения (valueType) и тип доступа (accessType).

Имя свойства (propertyName) берется непосредственно из дескриптора. Типа значения свойства (valueType) является результатом применения операции Type.forClass(Class<?> c) к типу значения с точки зрения JVM, то есть к классу, указанному в дескрипторе. Значение accessType является описанием способов доступа к данному свойству, поддерживаемых его реализацией; оно формируется по информации дескриптора и содержит признаки наличия возможных операция над свойством (скалярных и/или индексированных). Таким образом, объекты типа PropertyType, используемые в описаниях типов BeanVM, являются аналогами объектов типа PropertyDescriptor, используемых в описаниях классов JVM при их интроспекции.

Заметим, что операция Type.forClass(Class<?> c), как и механизм интроспекции JVM, обрабатывает любые java-классы, включая классы-массивы; они тоже представляются соответствующими типами BeanVM. Однако, типы BeanVM, реализация которых не унаследована из класса Bean, не являются ссылочными типами BeanVM (не являются компонентами BeanVM); они могут быть использованы только в качестве типов значений свойств у инстансов компонент. Компонентом-классом BeanVM (по определению) является только тип BeanVM, классом реализации которого служит JavaBeans-компонент, имеющий своим предком в дереве наследования предопределенный суперкласс Bean.

Для реализации инстансов всех компонент-классов суперкласс Bean предоставляет конструктор без аргументов, который неизбежно вызывается при инстанциировании класса реализации любого компонента-класса. Этот конструктор обеспечивает получение BeanVM-типа для инстанциируемого класса и реализацию инстанса этого типа. Получение типа по классу реализации происходит с привлечением загрузчика типов, поддерживающего отображение класса JVM в соответствующий тип-класс BeanVM. Реализация инстанса типа содержит постоянную ссылку на свой тип.

Далее происходит реализация внутреннего представления свойств данного инстанса в памяти BeanVM. Рассмотрим сперва «бесконтекстное» инстанциирование компонента-класса, которое соответствует его инстанциированию в JVM как «обычного» JavaBeans-компонента, каковым он является. Так инстанциировался бы этот компонент в стандартном контейнере BeanBox.

При таком инстанциировании мы должны получить готовый к работе инстанс компонента с

инициализированными значениями всех его свойств. Однако, стандартный механизм интроспекции не предоставляет нам всей необходимой для этого информации: в дескрипторах свойств нет информации о значениях, которыми они должны быть инициализированы. Такая инициализация обеспечивается только поведением конструкторов классов реализации. Мы могли бы потребовать явного предоставления этой информации, например, путем нестандартных соглашений о программировании JavaBeans-компонент, либо предоставлять явные BeanInfo-классы (умножая количество классов), либо предоставлять отдельные описания такой информации (в разных бытующих для этого форматах), либо наплодить нестандартные (с точки зрения JDK) аннотации, как это делают разные популярные библиотеки… Мы предпочитаем, чтобы программа сама умела выразить и сделать то, что ей положено.

Нам надо получить информацию о том, как должны быть инициализированы свойства у инстансов нашего компонента при инстанциировании его класса реализации вызовом его конструктора без параметров: дополнительно к объектам PropertyType(s), создаваемым на основании инстроспекции класса реализации, нам надо узнать, какими значениями соответствующих типов должны быть инициализированы свойства при создании инстанса компонента. Память для хранения их значений будет определяться не самим классом реализации инстанса компонента, а его суперклассом Bean. Для получения значений инициализации мы применяем ту же идею, что будем «в больших масштабах» использовать в дальнейшем: мы создадим инстанс прототипа, который сам проделает нужную инициализацию, а затем извлечем из него информацию, необходимую для уточнения типа нашего (бесконтекстного) компонента (в данном случае – значения инициализации его свойств).

Чтобы создать инстанс прототипа и позволить конструктору класса реализации инициализировать этот инстанс, суперкласс Bean предоставляет *внутренний* BeanAPI, которым пользуются все его наследники.

3.3 Внутренняя реализация интерфейса инстанса компонента

Под внутренней реализацией интерфейса инстанса компонента мы понимаем реализацию набора его свойств средствами предоставления типизированной памяти BeanVM.

Решение этой задачи обеспечивает инстанс типа BeanVM, создаваемый фабрикой, работающей внутри конструктора инстанса любого компонента - внутри конструктора суперкласса Bean. При этом фабрика, создающая внутреннюю реализацию инстанса компонента учитывает контекст его инстанциирования (наличие и тип контейнера, в котором происходит инстанциирование).

Поскольку реализации инстансов компонент-классов и инстансов составных компонент имеют общий суперкласс Bean, создания инстансов обеспечиваются сходным образом.

3.3.1 Компонентная реализация памяти инстансов компонент

В принципе, можно использовать идею компонентности, начиная с уровня ячеек типизированной памяти BeanVM, чтобы с их помощью обеспечить реализацию отдельных свойств для инстансов компонент.

При этом можно использовать простейший тип – «Типизированная переменная» (TypedVariableType), инстансы которого представляют собой «ячейки памяти», способные хранить значения заданного типа T, что контролируется динамически при выполнении операций записи. Помимо такого присваивания, имеются операции чтения значения и операции регистрации/дерегистрации объектов, оповещаемых при изменениях значения. Такие типизированные переменные можно рассматривать как динамический аналог параметризованного типа Variable<T> с тремя операциями: write (W), read (R) и bind (B) – для реализации связываемых свойств (bound properties). Для хранения значений

индексированных свойств (и/или значений, чей тип значений – массив), добавляются операции indexed-write (w) и indexed-read (r); мы ограничиваемся поддержкой одномерных массивов в качестве значений реализуемых свойств.

Чтобы создать новый тип для создания экземпляров переменных со значениями типа T («компонент-фабрику» создания типов Variable<T>), нам надо иметь объект-прототип (типа TypedVariablePrototype), который обладает функциональностью типизированной переменной Variable<T> с операциями (W,R,B), где операция записи динамически контролирует тип T записываемого значения, а также – позволяет менять у себя текущий тип T на новый T' с «одновременной заменой» текущего значения типизированной переменной на значение нового типа T', используемое по умолчанию.

Мы используем общее правило получения «глобального значения по умолчанию» (global default value) для любого типа T. Все типы BeanVM подразделяются на *типы-значения* (*value types*) и *ссылочные типы* (*reference types*); любой ссылочный тип имеет глобальное значение по умолчанию равное null, и каждый тип-значение до своего использования регистрирует себя в глобальной таблице таких типов BeanVM вместе со своим глобальным значением по умолчанию. Типы-значения, реализованные java-классами, могут регистрировать себя в этой таблице при выполнении своих статических инициализаторов.

Тип TypedVariablePrototype может быть реализован как тип-класс BeanVM, имеющий свойство “valueType” и свойство “value”, с указанной выше логикой реакции на изменения их значений. Инстанс такого компонента-прототипа может быть легко преобразован в нужный тип типизированной переменной BeanVM - TypedVariableType (в нем присутствует вся необходимая для этого информация). Заметим, что объектом-прототипом могла бы быть сама переменная с записанным в ней типизированным значением (без свойства “valueType”), если бы по нему можно было узнать его тип значения (valueType), чему препятствует значение null (мы здесь не вдаемся в дискуссию об этом).

Динамический контроль типа позволяет автоматически создавать типизированные переменные на этапе исполнения – даже для тех типов значений, которые сами будут динамически генерироваться. Для типов значений, реализованных заранее известными классами, можно использовать JavaBeans-компоненты со свойством “value”, имеющими нужные JVM-типы значений (включая примитивные) непосредственно в реализациях методов доступа. Готовые классы реализации таких JavaBeans-компонент можно искать и подгружать динамически, получая их имена с помощью дополнительной мета-информации (соглашений об именовании). Эту возможность можно рассматривать как средство оптимизации работы BeanVM методами кодогенерации в JVM.

Такой «полностью компонентный» подход к организации памяти BeanVM, начиная с уровня отдельных типизированных ячеек памяти, был опробован реализован в предшествующей версии проекта [7]. Он является избыточным («чрезмерно компонентным») потому, что мы имеем дело с компонентами, обладающими постоянными, фиксированными при их создании, наборами свойств известных типов. Мы рассматриваем компонентную модель, где элементарным (атомарным, неделимым) компонентом является компонент специального вида с определенным набором свойств, а не отдельная его составная часть (отдельное свойство). Мы имеем дело с «молекулами», а не с «атомами», и всегда знаем количество свойств и их типы. Внутренняя реализация типизированной памяти BeanVM средствами суперкласса Bean скрыта внутри него (допуская оптимизацию).

3.3.2 Внутренний интерфейс реализации инстансов

Внутренний интерфейс для реализации доступа инстансов компонент к их свойствам, предоставленный в базовом классе Bean, выглядит следующим образом:

```
protected final Object get(int propertyIndex); // операция чтения
protected final Object get(int propertyIndex, int elementIndex); // индексированное чтение
```

```
protected final void set(int propertyIndex, Object newValue); // операция записи
protected final void set(int propertyIndex, int elementIndex, Object newValue); // индексированная запись
```

Автор реализации компонента (класса, наследованного из класса Bean, которому доступны эти операции), пользуется этими методами при реализации внешних методов доступа к свойствам инстанса компонента.

Например, для реализации свойства “foo” со значением примитивного типа float можно в классе реализации компонента определить методы следующим образом:

```
public void setFoo(float newValue) {set(FOO_INDEX, newValue);}
public float getFoo() {return (float) get(FOO_INDEX);}
```

Здесь подразумевается получение propertyIndex по propertyName статическим инициализатором класса:

```
static final int FOO_INDEX = Type.getPropertyIndex(“foo”);
```

Эта операция реализуется с использованием информации, известной на этапе создания типа по классу его реализации.

Заметим, что в данном примере реализации свойства “foo” скрыты неявные операции boxing/unboxing для примитивного типа float, но не скрыто явное нисходящее преобразование (downcast) к типу, возвращаемому при выдаче значения свойства; оно необходимо для любых возвращаемых этим методом типов (при отсутствии специальной кодогенерации для реализации свойств).

По умолчанию свойство “foo” реализуется как «связываемое» (bound property), что соответствует использованию аннотации @BeanProperty (из современного стандартного Java API). Правомочность операций W и R обеспечится наличием соответствующих методов доступа свойства. Контроль типа при присваивании будет обеспечен динамическими средствами. Мы оставляем возможности многочисленных оптимизаций средствами кодогенерации, в том числе – динамической, методами Monkey Patching [8] с помощью разных имеющихся для этого библиотек, за рамками нашего обсуждения.

3.3.2 Бесконтекстное инстанциирование

Вернемся к инстанциированию компонента-типа. Реализация инстанса включает в себя выделение памяти для хранения значений свойств; инициализация обеспечивает наличие в них начальных значений.

Например, если у нас есть компонент-тип “Bar” со свойством “foo”, реализованный классом Bar.class, то операция Type.forClass(Bar.class) вернет нам его тип Bar, где будет тип свойства, содержащий:

- “foo” – propertyName (имя свойства);
- Type.forClass(float.class) – property valueType (тип значения свойства);
- (W,R,B) – property accessType (writable, readable, bound – права доступа свойства).

При реализации инстанса такого бесконтекстного типа, мы отведем для хранения значения свойства “foo” переменную, обеспечив выполнение для нее указанных операций. Автор JavaBean-компонента Bar.class может инициализировать свойство “foo” значением, например, 1.0F (если глобальное значение по умолчанию 0.0F для типа float не годится).

При создании компонента-типа (типа по классу) создается его первый – служебный – инстанс, который является инстансом-прототипом (будем называть его *протоинстансом*) всех последующих инстансов этого типа. Протоинстанс при создании получает внутреннюю реализацию всех своих свойств, но так, что каждое из этих свойств реализуется переменной с максимально возможным типом доступа (accessType(s)) для своего типа значения valueType (для скалярных типов свойств это будет (W, R, B), для индексированных это будет (W, R, B,

w, r). В качестве начальных значений свойств протоинстанса берутся глобальные значения по умолчанию для типов значений свойств. Все это происходит в контексте выполнения конструктора суперкласса Bean, который неизбежно вызывается перед вызовом конструктора любого компонента. После создания протоинстанса суперконструктором Bean() дорабатывает конструктор исходного класса компонента, который получает возможность уточнить начальные значения своих свойств, пользуясь максимальными правами доступа внутренней реализации протоинстанса и специальными операциями инициализации, которые реализация протоинстанса для этого предоставляет:

```
protected final void initPropertyValue(int propertyIndex, Object initValue);
protected final void initPropertyValue(int propertyIndex, int elementIndex, Object initValue);
```

После инициализации протоинстанса конструктором компонента с помощью этих операций происходит окончательное формирование BeanVM-типа: типы свойств дополняются уточненными начальными значениями инициализации (отличными от глобальных значений по умолчанию). Этими итоговыми значениями протоинстанса будут инициализироваться все последующие инстансы (бесконтекстного) компонента – уже без помощи его конструктора: *реализации указанных операций инициализации всех инстансов типа, кроме протоинстанса, являются пустыми операциями*.

Мы видим, что даже внутренняя реализация бесконтекстного инстанса компонента-типа зависит от контекста инстанциирования самой этой реализации: она может быть реализацией протоинстанса или инстанса готового типа (что влияет на выполнение операций).

Разумеется, права доступа свойств у инстансов типа (в отличие от протоинстанса) определяются (уточняются в контексте использования) так, как указано в их реальных типах.

При «самореализации» инстансов по их типам используются правила отведения памяти в зависимости от действующих прав доступа: если свойство допускает операции связывания или записи (B, W и/или w), то потенциально изменяемое значение хранится в переменной (RAM BeanVM); иначе значение хранится в константной ячейке (ROM BeanVM).

Компонент-класс BeanVM реализован как «обертка» своего JavaBean-класса, но инстанс этого класса реализации, является «оберткой» инстанса своего BeanVM-типа, выполняющего внутреннюю реализацию инстанса компонента-класса. Это иллюстрирует рис. 2.

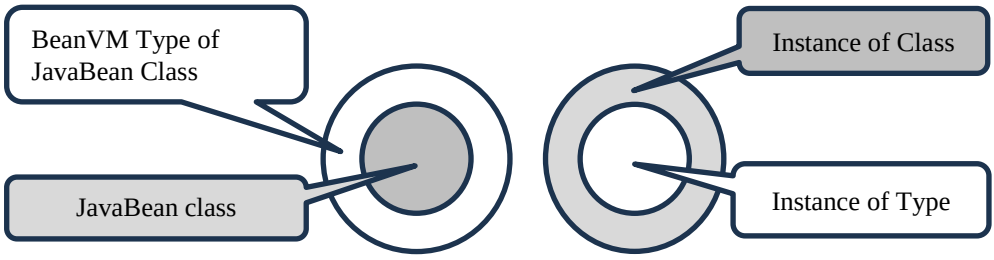


Рис. 2. Отношения вложенности типов BeanVM, классов JVM и их инстансов.
Fig. 2. Nesting relationships of BeanVM types, JVM classes and their instances.

При программировании JavaBean-класса реализации BeanVM-компонента использование внутреннего BeanAPI выглядит естественно и не требует других, специальных средств предоставления значений инициализации.

Вместе с тем, есть и альтернативный способ доопределения компонента-класса значениями для инициализации свойств, не связанный с использованием конструктора. Можно просто передать такие значения инициализации в статическом инициализаторе класса реализации компонента. Для этого есть метод доопределения типа, принимающий параметром Map<String, Object>, с помощью которого статический инициализатор класса реализации

типа может уточнить своему типу BeanVM значения инициализации поименованных в нем свойств.

Тем не менее, подход с использованием протоинстанса полезен тем, что он уже на уровне элементарных компонент дает пример преобразования инстанса прототипа в тип.

3.3.3 Контекстно-зависимое инстанциирование

Смысл создания компонент заключается в многократном использовании их инстансов при разных обстоятельствах. Мы допускаем, что инстанс класса Bar из приведенного ранее примера будет использоваться там (в таком контексте), где изменений значения свойства “foo” вообще быть не должно, а само значение свойства “foo” должно быть равно 2.0F.

Мы хотим использовать инстансы компонент-классов в разных контекстах и предполагаем возможность настройки реализации инстансов на контекст их использования. Это означает, что при *инстанциировании одного и того же класса* реализации компонента нам надо уметь *инстанциировать разные типы BeanVM*, инстансы которых являются внутренними реализациями инстансов класса. Фабрика создания внутренних реализаций должна учитывать контекст инстанциирования.

По определению компонент-классов, они, являясь JavaBeans-компонентами, могут инстанциироваться в JVM только вызовом их конструктора без аргументов (без возможности передать ему контекст инстанциирования). Тем не менее, в контексте вызова такого конструктора фабрика создания внутренней реализации инстанса (в суперконструкторе Bean()) может получить эту информацию с помощью операций BeanVM (обращения к скрытому стеку вызовов BeanVM, доступному для базового класса Bean). Этот прием позволяет конструктору компонента создать его внутреннюю реализацию инстансом типа BeanVM с учетом информации о контексте инстанциирования и предоставить при этом разные контекстно-зависимые возможности.

Прием с использованием неявной передачи контекстной информации при инстанциировании (через скрытый стек BeanVM) можно сравнить с инстанциированием нестатических вложенных классов (*inner*) языка Java, но в них передача неявного параметра (ссылки на охватывающий объект) обеспечивается соответствующей кодогенерацией при компиляции; мы используем динамический аналог.

Соответственно, BeanVM предоставляет разные операции инстанциирования компонент. Операция бесконтекстного инстанциирование компонент-классов не отличается от способов инстанциирования JavaBeans-компонент (каковыми они являются).

Дополнительно, имеются операции инстанциирования в «контейнерах» – инстансах специальных типов BeanVM, внутри которых (в контексте которых) создаются инстансы BeanVM-компонент. Эти операции имеют вид:

<container>.instantiate (ComponentType type) --> <instance>,

где <container> - инстанс компонента-контейнера, type – инстанциируемый компонент (тип) BeanVM, и <instance> – получаемый инстанс (объект типа type). Заметим, что так могут инстанциироваться только типы-компоненты; типы, не являющиеся компонентами (*типы-значения*), инстанциируются поведением самих компонент (средствами JVM), и их инстансы видимы в BeanVM только как значения свойств инстансов компонент.

Компоненты определяют максимальную функциональность своих инстансов *a-priori*, до их конкретного использования, где она не всегда необходима. Классические JavaBeans-компоненты на этом останавливаются и позволяют настраивать свои инстансы только путем изменения значений свойств (вызовами имеющимися для этого методов).

Наши компоненты-классы могут настраивать свои инстансы на контекст их использования благодаря их внутренней реализации инстансами BeanVM-типов, зависящими от контекста инстанциирования, без изменения класса реализации. Фабрика внутренних реализаций

инстансов компонент-классов (в суперклассе Bean) понимает контекст инстанциирования, а все операции внешнего интерфейса инстанса компонента делегируют к методам инстанса внутренней реализации. Инстанс JavaBean-класса при выполнении конструктора суперкласса Bean получает final-ссылку “thisImpl” на реализующий его инстанс соответствующего типа BeanVM (как показано на рис. 2).

Поскольку класс компонента всегда неизменен, а поведение инстансов определяется в нем, контекстно-зависимое изменение их функциональности может происходить только «в меньшую сторону», за счет того, что для данного контекста некоторая имеющаяся функциональность оказывается «незадействованной». Это и определяется с помощью протоинстанса (инстанса прототипа), в котором можно указать избыточные для данного контекста возможности (операции над свойствами) и убедиться в этом на практике.

Фабрика, которая обеспечивает внутреннюю реализацию инстансов компонент, реализует их свойства с учетом различных факторов, в зависимости от которых доступ к свойству может декорироваться. Такими факторами являются:

- тип доступа к свойству (в зависимости от него обеспечивается память для свойства);
- тип значения свойства (типы значений могут сами доопределять правила поведения методов доступа и правила валидации значений, нарушения которых приводят к соответствующим исключительным ситуациям; доступ к значениям-массивам JVM реализуется с копированием этих значений; можно управлять возможностью присваивания значения null, и т.п.);
- контекст инстанциирования (реализации свойств инстансов-прототипов и инстансов составляющих типов отличаются);
- контекст использования инстанса (влияет на прототипирование составляющих типов и их последующее инстанциирование).

3.4 Составные компоненты

Составные (composed) компоненты – это *динамически определяемые типы-компоненты*, которые не являются представителями классов своих реализаций в BeanVM, но создаются в динамике с помощью компонент, инстансы которых которые создаются, агрегируются и настраиваются в контексте инстанса специального контейнера-прототипа, после чего он преобразуется в составной тип (composed-type).

Как и компоненты-классы, composed-компоненты – это типы BeanVM, которые уникально именуются и предоставляют интерфейс к своим инстансам в терминах свойств (теми же средствами): реализации их инстансов тоже обеспечиваются базовым классом Bean. Поведение инстансов составных компонент определяется совместным поведением инстансов составляющих их типов.

3.4.1 Прототипирование составных типов

Для динамического определения составных компонент используется компонент-класс ComposedBeanPrototype - прототип составного типа (*мета-компонент* создания компонент).

Инстанс ComposedBeanPrototype – это контейнер инстансов любых компонент BeanVM. Он является для них *контекстом инстанциирования*. Инстансы компонент, создаваемые в нем, используются для *агрегирования* и настройки *работоспособного* составного протоинстанса. При этом они играют роль прототипов, из которых будут произведены составляющие типы (*composing-types*) композиции – составного типа (*composed-type*), получаемого в результате преобразования инстанса контейнера-прототипа в тип.

Создавая инстансы компонент в контексте инстанса контейнера-прототипа, мы хотим уметь их настраивать и использовать совместно для обеспечения функционирования протоинстанса, составляемого из них. Для этого при инстанциировании компонента в таком

контексте предоставляются дополнительные возможности манипуляций с инстансом, выходящие за рамки возможностей бесконтекстно инстанциированного компонента. Эти возможности обеспечивает специальный объект – «обертка» (wrapper) создаваемых инстансов, предоставляемый контейнером-прототипом. Он позволяет реализовать:

- доступ к свойствам самого инстанса (тем самым прототипируются контекстно-зависимые уточнения значений инициализации для будущего составляющего типа);
- понижение типа доступа к свойству относительно заданного в типе компонента (при прототипировании мы хотим уметь «сознательно отказываться» от избыточных в данном *контексте использования* возможностей инстанса компонента; мы не изменяем исходный бесконтекстный компонент, но «декорируем» реализацию доступа к значениям его свойств переменными полями их текущих прав; это позволяет прототипировать контекстно-зависимые понижения типов доступа к свойствам для будущего составляющего типа;
- замену внутренней реализацию свойства инстанса на реализацию однотипного (по типам значения и доступа) свойства другого инстанса, имеющегося в их общем инстансе контейнера-прототипа (это средство «разделения» реализаций свойств мы уточним ниже).

3.4.2 Определение свойств составных компонент

Составной (composed) компонент определяет конкретный набор типов свойств (возможно пустой) для взаимодействия с его инстансами. В реализации – это массив объектов – типов свойств, которые используются и для определения типов свойств компонент-классов. Способом получения типа набора свойств является преобразование его прототипа в тип. Для этого надо:

- создать инстанс прототипа (компонента-класса PropertySetPrototype);
- заготовить набор типов значений свойств будущего составного компонента;
- сформировать Map<String, Type> с именами и типами значений свойств;
- присвоить этот объект в свойство “valueTypesMap” инстанса PropertySetPrototype; поведение прототипа создаст и предоставит протоинстанс, у которого будут иметься свойства с указанными именами и типами значений; они будут инициализированы глобальными значениями по умолчанию своих типов значений и предоставлять максимально возможный доступ; созданный объект-протоинстанс предоставляется как выходное значение свойства “protoinstance” инстанса прототипа;
- инициализированные значения свойств можно изменить (либо с помощью свойства прототипа “initValuesMap”, куда надо передать Map<String, Object> с задаваемыми значениями, которые будут использованы только для соответственно именованных свойств протоинстанса, либо доступными операциями записи значений);
- последним шагом является указание о понижении прав доступа для свойств протоинстанса; для этого используется свойство “accessTypesMap” со значением Map<String, AccessType> с пониженными типами доступа у именованных свойств.

Последний шаг указывает прототипу о завершении настройки протоинстанса и стимулирует выдачу результата – компонента PropertySetType (типа набора свойств) – в качестве значения одноименного выходного свойства его прототипа. Этот шаг является обязательным (если понижения прав доступа не требуются, надо просто передать пустой Map-объект).

Типы BeanVM предоставляются операцией Type.forName(String typeName). Имя компонента-класса заимствуется из его класса реализации. Имя составного компонента определяются значением свойства “name” инстанса его прототипа (уникальность имен обеспечивается загрузчиками типов, контролирующими их создание и именование).

3.4.3 Прототипирование взаимодействия инстансов компонент

Составной прототип должен реализовать функциональность протоинстанса с внешним интерфейсом к нему в терминах набора свойств.

Внешний интерфейс любого инстанса компонента реализуется инстансом BeanVM-типа PropertySetType с реализацией JVM-классом PropertySet (наследником класса Bean), предоставляющим реализацию конкретного набора свойств инстанса компонента. Эти свойства «соединяются» с однотипными свойствами других инстансов компонент, связывая интерфейс с реализацией.

Популярным способом взаимодействия между однотипными свойствами разных инстансов JavaBeans-компонент являются сигналы PropertyChangeEvent(s), передаваемые от связываемых свойств (bound properties); эти средства сохраняются. Однако, реализация *двусторонних связей* между инстансами компонент с помощью таких событий от свойства источника к однотипному по значению свойству приемника является громоздкой, сопровождается переписью значений и не решает задачу, так как работает только для связываемых свойств.

Поэтому имеется механизм реализаций свойств «с общей памятью», где значения не надо переписывать, обращаясь к ним по ссылкам (аппаратная аналогия: контакты элементов можно соединять посредством проводов, но можно их соединить и непосредственно).

Смысл инстансов типа PropertySetType заключается в предоставлении ими определенного набора реализаций свойств, которые могут разделяться другими инстансами компонент. Инстансы, реализующие наборы свойств, не имеют определенного в их типе поведения, но позволяют использовать поведения, определенные реализациями компонент, инстансы которых разделяют реализации своих свойств с ними. Необходимым условием разделения реализации свойств является их однотипность – совпадение типов значений и доступа. Инстансы компонент, использующие разделяемые реализации свойств, обращаются к их значениям и реагируют на их изменения так же, как при использовании своих собственных реализаций свойств. Использование разделяемых свойств не отражается на механизме реализации связанных свойств, так как регистрация/дерегистрация слушателей связана с самим объектом свойством, а не с памятью, хранящей значение (свойство реализуется как контекст использования типизированной ячейки памяти, а не она сама).

При создании инстанса компонента в контексте инстанса контейнера-прототипа контекст инстанциирования предоставляет возможность доступа к другим имеющимся в этом контексте инстансам. Их можно найти благодаря регистрации всех инстансов под их локально-уникальными именами внутри таблицы именованных объектов инстанса контейнера-прототипа. Специальная операция позволяет инстансу в контейнере прототипа в качестве реализации своего свойства использовать реализацию свойства из инстанса другого компонента (набора свойств) этого же контейнера. Операция выполняется успешно только при совпадении типов значений и доступа используемого (*разделяемого*) свойства и использующего (*разделяющего*) свойства. При этом одно разделяемое свойство может разделяться многими разделяющими свойствами от разных инстансов компонент, список которых поддерживается для оповещения всех инстансов при изменении общего значения кем-либо из них.

С точки зрения инстанциирования это означает, что внутренняя реализация инстанса в контексте контейнера прототипа допускает последующую замену реализации его свойства - с использования контекста его собственной памяти для хранения значения на использование контекста памяти разделяемого свойства другого инстанса из того же контейнера (который должен быть известен как контекст инстанциирования).

Контекстно-зависимая настройка типов доступа может производиться только для свойства, которое разделяется; разделяющие свойства всегда имеют тип доступа разделяемого

свойства. Разделение свойств возможно только у инстансов PropertySetType; это связано с их ролью в реализации поведения инстансов составных типов.

Информация о реализациях свойств при прототипировании переносится в описания составляющих типов составного типа, получаемого из прототипа; она используется при инстанциировании составляющих типов в контексте инстанса составного типа, когда создается он.

3.4.4 Прототипирование поведения

Взаимодействие инстансов компонент происходит в терминах операций с их свойствами и подразумевает изменения их значений. Стандартным способом организации совместного поведения инстансов JavaBean-компонент является создание графа распространения событий между ними, получаемых от «связываемых свойств» (bound properties). Механизм распространения событий типа PropertyChangeEvent обеспечивает реализацию поведения с помощью «каскадов событий», которые завершаются даже при наличии циклов в графе их распространения. При этом явной зависимости результатов общего поведения инстансов компонент от контекстов их использования при агрегировании может не быть.

Реализация поведение определяется типом контейнера, в котором моделируется поведение, и типами используемых в нем компонент. Есть ряд важных задач моделирования, где контексты использования инстансов компонент требуют учета. К ним относятся задачи моделирования реальных объектов: компьютерная графика, программирование GUI (изначально основная область применения JavaBeans-компонент, из-за чего пакет java.beans оказался в модуле java.desktop), и другие.

Контексты использования инстансов компонент возникают, когда одни инстансы компонент присваиваются в качестве значений свойств других инстансов, допускающих такое присваивание (*setter dependency injection*). При таких присваиваниях возможно создание направленного графа ссылок с вершинами (узлами) из инстансов компонент и со связями от узла, обладающего свойством со ссылочным типом значения, к узлу, который является таким значением.

Для учета контекстных зависимостей часто используются деревья, что является ограничением, затрудняющим разделение данных в узлах разными контекстами. С другой стороны, наличие циклов в графах затрудняет учет контекстов использования узлов.

Представление контекстных зависимостей в виде направленного ациклического графа (Directed Acyclic Graph - DAG) позволяет естественно переиспользовать общие данные узлов. Обход (траверсирование) такого графа позволяет получить информацию о контекстах использования узлов на стеке траверсирования графа. Такой вариант является расплатой временем траверсирования (с многократным построением стека) за экономию памяти и затрудняет параллельное использование модели данных.

При реализации контейнеров и компонент, которые предполагают использование контекстных зависимостей, в BeanVM используется «компромиссный» вариант представления инстанса контейнера: одновременно с DAG из инстансов компонент, который создается setter’ами свойств ссылочных типов и допускает использование одного инстанса несколькими другими, мы строим *дерево контекстов* использования узлов этого графа.

Узлы дерева контекстов хранят контекстно-зависимую информацию и не требуют многократного траверсирования (мы расплачиваемся памятью дерева за экономию времени траверсирования графа). Изменения топологии графа и соответствующая корректировка дерева контекстов происходят гораздо реже, чем возникает необходимость в использовании контекстно-зависимой информации (например, для графического отображения). Кроме того, наличие дерева контекстов узлов графа упрощает валидацию его ацикличности.

3.4.5 Взаимосвязь прототипа и протоинстанса

В терминах графа (DAG), узлами которого являются инстансы компонент, и дерева их контекстов можно пояснить связь между прототипом и его протоинстансом. Составной прототип представляется графом, который строится с помощью присваиваний свойствам значений ссылочных типов (setter dependency injection). Протоинстанс (такого прототипа) представляется деревом контекстов узлов графа, на котором отражается и с помощью которого реализуется поведение протоинстанса. При прототипировании эти структуры данных строятся, настраиваются и функционируют одновременно.

При инстанциировании типа, реализация которого до его инстанциирования получена из прототипа, инстанс создается уже при наличии постоянного (immutable) описания типа, где можно хранить автоматически обобщаемую всеми инстансами информацию, что открывает возможности для значительной оптимизации ресурсов памяти и времени.

В терминах DAG и его дерева контекстов можно выразить и внутреннюю организацию инстансов элементарных компонент - реализацию их свойств в базовом классе Bean (хотя она скрыта от пользователя). Роль базовых компонент при этом играют типизированные переменные. Их инстансы образуют прототип. Контексты инстансов типизированных переменных - суть свойства, с помощью которых реализуются методы доступа к значениям в узлах прототипа. Протоинстанс является корнем дерева, содержащего свойства, и внутренней реализацией прототипа, который является контейнером узлов графа. Инстанс прототипа содержит корень дерева (протоинстанс), и протоинстанс, как корень дерева контекстов, имеет ссылку на контейнер графа; просто и граф, и дерево представлены своими простейшими вариантами («кустами»). Уже для элементарных компонент такая реализация позволяет автоматически переиспользовать постоянные (immutable) значения свойств.

4. Обсуждение

Мы рассмотрели основы эволюционного подхода к развитию компонентной модели JavaBeans в направлении добавления к ней динамических средств создания компонент. Этот подход позволяет расширить возможности компонентной модели путем добавления новых наборов компонент, являющихся JavaBeans-компонентами, для прототипирования и создания новых типов-компонент в динамике. Описание всех аспектов предложенного подхода выходит за рамки данной статьи, в которой изложен общий подход, основные понятия и связанные с ними направления реализации.

Компоненты – это *типы деталей*, которые достаточно *универсальны* (не ограничены контекстом создания) и сконструированы *заранее* (до создания самих деталей) для определения *конечных продуктов* и/или *новых компонент*. При создании компонент в динамике *без кодогенерации* требуется расширенное понятия типа, включающее в себя типы, создаваемые на этапе исполнения (at runtime).

Сборка с применением компонент предполагает использование *самих деталей* (инстансов имеющихся типов) в некотором специальном «ящике» – *прототипе* (он сам – деталь своего типа). Там создается *агрегат* из других деталей по определенным для этого правилам. Эти правила определяет компонентная модель; реализуют правила ее компоненты, инстансами которых могут быть контейнеры-прототипы, внутри которых происходит агрегация, и компоненты, инстансы которых агрегируются в них. Соблюдение этих правил «заложено в генах» компонент (в их типах). Корректное выполнение правил реализуется (воплощается) в их инстансах с помощью наследования ими общей для них реализации.

В прототипе возникает (составной) образец деталей нового типа. Этот образец реально работоспособен, он настраивается и проверяется. Из него можно извлечь состояние агрегации, где все элементы (сама деталь и ее связи с другими деталями в агрегации) будут служить образцами для определения композиции – *составного типа* и его *составляющих типов*, показывающих, как надо применять (создавать и использовать) такие детали.

Это *не есть копирование образца*: составляющие типы учли контекст использования своих деталей и избавились от их ненужной универсальности. Это может дать большую экономию ресурсов (памяти и времени) при применении нового типа (по тем же правилам).

5. Распространенные подходы

Парадигма компонентно-ориентированного программирования давно и широко обсуждается в литературе. При этом ее практическое использование нельзя назвать повсеместным, что обусловлено спецификой предметных областей, решаемых задач и используемых средств.

За годы развития Java-платформы накоплен большой опыт использования компонент при создании как локальных приложений, так и больших распределенных систем. Почти везде при этом используется термин «Bean», но далеко не всегда следуют его определению в [4].

Начиная с первых Java-технологий для серверных приложений, наметилась тенденция использования компонентных моделей, ориентированных на использование компонент в специфических контейнерах (таких, как модель Enterprise JavaBeans [9]). В них допускается применение JavaBeans-компонент, играющих вспомогательную роль. Этот подход используют и современные технологии, основанные на Spring Framework [10], где понятие «Bean» имеет свой смысл, отличный от его определения в [4].

JavaBeans-компоненты используются в библиотеках создания GUI, в том числе в JavaFX [11], где есть набор специфических реализаций свойств компонент, но нет выхода за рамки использования скомпилированных типов.

Все эти технологии прошли многолетний путь: от программной конфигурации составных частей, к использованию для этого специальных форматов (XML-подобных) и к применению многочисленных специальных аннотаций (без явного определения компонентной модели).

Они не дают примеров использования стандартных компонент для получения «себе подобных» без кодогенерации, стандартными средствами самих компонент, и не предоставляют средств оптимизации (контекстно-зависимой), упомянутых ранее. Эти средства связаны с наличием общего базового класса реализации инстансов компонент в BeanVM, тогда как, например, в идеологии Spring Framework подчеркивается отсутствие общего суперкласса, отличного от класса java.lang.Object (и выхода за рамки традиционных средств Java-программирования).

В работе предлагается подход к реализации новых возможностей в рамках исходной компонентной модели, которую можно дополнять новыми компонентами, снимающими необходимость в кодогенерации и допускающими реализацию специфических контейнеров и их компонент, а не наоборот (когда специфические контейнеры по-своему определяют компонентную модель).

Возможность кодогенерации остается как средство оптимизации. Методы кодогенерации могут переводить *уже работоспособный составной компонент* в эквивалентный компонент-класс (из BeanVM в JVM, что напоминает JIT-компиляцию, которая сработает потом).

6. Заключение

В настоящее время проект состоит из пилотных проектов, где опробованы и протестированы основные рассмотренные в работе решения. Предполагается объединение этих проектов в программный продукт – библиотеку для прикладного использования. Необходимо также предоставить инструмент для визуального манипулирования компонентами, поддерживающий новые возможности, который может стать средством компонентного программирования.

Список литературы / References

[1]. Douglas McIlroy. Mass-produced software components. Электронный ресурс. https://www.cs.dartmouth.edu/~doug/components.txt (дата обращения 22.08.2025).

[2]. A.J.A.Wang, K.Qian, Component-oriented programming, John Wiley & Sons, Inc., 2005.

[3]. Kung-Kiu-Lau, Zheng Wang (2006) A Survey of Software Component Models (second edition), School of Computer Science, The University of Manchester, Preprint Series, CSPP-38.

[4]. JavaBeans(TM) Specification. Электронный ресурс. https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/
Архивировано:
https://web.archive.org/web/20210416231451/https://www.oracle.com/java/technologies/javase/javabeans-spec.html (дата обращения 22.08.2025).

[5]. VRML – The Virtual Reality Modeling Language. Электронный ресурс. https://www.web3d.org/documents/specifications/14772/V2.0/ (дата обращения 22.08.2025).

[6]. Проект Babylon. Электронный ресурс. https://openjdk.org/projects/babylon/ (дата обращения 22.08.2025).

[7]. E.Grinkrug. A Framework for Dynamical Construction of Software Components. A.K.Petrenko and A.Voronkov (Eds.): PSI 2017, LNCS 10742, pp. 163-178, 2018. https://doi.org/10.1007/978-3-319-74313-4_13.

[8]. N. Frankel. Monkey-Patching in Java. Электронный ресурс. https://dzone.com/articles/monkey-patching-in-java. (дата обращения 22.08.2025).

[9]. Enterprise JavaBeans Technology. Электронный ресурс. https://www.oracle.com/java/technologies/enterprise-javabeans-technology.html (дата обращения 22.08.2025).

[10]. Spring Framework. Электронный ресурс. https://spring.io/projects/spring-framework (дата обращения 22.08.2025).

[11]. JavaFX Documentation. Электронный ресурс. https://openjfx.io (дата обращения 22.08.2025).

Информация об авторах / Information about authors

Ефим Михайлович ГРИНКРУГ – кандидат технических наук, старший научный сотрудник Института системного программирования им. В.П. Иванникова РАН с 2024 года. Сфера научных интересов: операционные системы, компонентно-ориентированное программирование, компьютерная графика, беспроводные сенсорные сети.

Efim Mikhailovitch GRINKRUG – Cand. Sci. (Tech.), Senior Researcher at the Ivannikov Institute for System Programming of the RAS since 2024. Research interests: operating systems, component-oriented programming, computer graphics, wireless sensor networks.

DOI: 10.15514/ISPRAS-2025-37(6)-3

Сравнение объектно-ориентированного и процедурно-параметрического полиморфизма

П.В. Косов, ORCID: 0000-0002-9035-312X <pvkosov@hse.ru>
А.И. Легалов, ORCID: 0000-0002-5487-0699 <alegalov@hse.ru>

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

Аннотация. Динамический полиморфизм часто применяется в ситуациях, связанных с определением и обработкой альтернативных ситуаций в процессе выполнения программ, обеспечивая гибкое расширение ранее написанного кода. Он широко используется в статически типизированных языках объектно-ориентированного программирования за счет совместного применения наследования и виртуализации. Языки программирования Go и Rust, также поддерживают динамический полиморфизм, используя для его реализации статическую утиную типизацию. Еще один подход предлагает процедурно-параметрическая парадигма программирования, обеспечивающая гибкое эволюционное расширение как вариантных данных, так и обрабатывающих их функций, включая ситуации, связанные с множественным полиморфизмом, которые возникают при реализации мультиметодов. В работе проводится сравнение возможностей динамического полиморфизма объектно-ориентированной и процедурно-параметрической парадигм, по поддержке гибкой разработки программного обеспечения. Сопоставляются базовые техники, обеспечивающие расширение функциональности программ, рассматриваются особенности реализации паттернов проектирования.

Ключевые слова: язык программирования; компиляция; процедурно–параметрическое программирование; полиморфизм; эволюционная разработка программного обеспечения; гибкая разработка программного обеспечения.

Для цитирования: Косов П.В., Легалов А.И. Сравнение объектно-ориентированного и процедурно-параметрического полиморфизмов. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 43–58. DOI: 10.15514/ISPRAS–2025–37(6)–3.

Comparison of Object-Oriented and Procedural-Parametric Polymorphism

P.V. Kosov ORCID: 0000-0002-9035-312X <pvkosov@hse.ru>
A.I. Legalov ORCID: 0000-0002-5487-0699 <alegalov@hse.ru>
Higher school of Economics, National research University,
20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. Dynamic polymorphism is widely used in situations involving the identification and processing of alternatives during program execution. Dynamic polymorphism allows to flexibly expand programs without changing previously written code. It is widely used in statically typed object-oriented programming languages by combining inheritance and virtualization. The programming languages Go and Rust also provide support for dynamic polymorphism, using static duck typing to implement it. Another approach to implementing dynamic polymorphism is offered by the procedural-parametric programming paradigm, which at the same time provides direct support for multimethods and flexible evolutionary expansion of both alternative data and their processing functions. The paper compares the capabilities of object-oriented and procedural-parametric paradigms to support agile software development. The basic techniques that ensure the expansion of the functionality of programs are compared. The features of the implementation of design patterns are considered.

Keywords: programming language; compilation; procedural-parametric programming; polymorphism; evolutionary software development; agile software development.

For citation: Kosov P.V., Legalov A.I. Comparison of object-oriented and procedural-parametric polymorphism. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 43-58 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-3.

1. Введение

При разработке программного обеспечения учитываются различные критерии качества. Расширение программы без изменения ранее написанного кода или с его минимальными изменениями является одним из них. Его важность обуславливается как неполным знанием об окончательной функциональности программ во время их создания и начальной эксплуатации, так и необходимостью ускорить выход продуктов на рынок за счет реализации только базового набора функций с последующим его наращиванием. В подобных ситуациях добавление новых конструкций, изменяющих уже написанный код, зачастую ведет к появлению неожиданных ошибок и непредсказуемому поведению.

Интерес к эволюционному расширению программ возник достаточно давно. Например, в работе [1] рассмотрена технология вертикальных слоев, выстраиваемых на основе выделения функций, расширяющих «обедненную» версию программы. Была предложена практическая реализация вертикального слоения в виде концепции сосредоточенного описания рассредоточенных действий. В работах [2-6], наряду с более детальным анализом механизмов реализации технологии вертикального слоения, даны технические рекомендации по ее реализации.

Эволюционное расширение во многом реализуется за счет динамического связывания программных объектов, что обеспечивает гибкое изменение структуры программы во время выполнения. При этом могут использоваться как явная, так и неявная обработка подключаемых альтернативных объектов. Последняя определяет динамический полиморфизм. Он отличается от статического полиморфизма, при котором выявление альтернатив осуществляется во время компиляции.

Изначально реализация динамического полиморфизма, была предложена для объектно-ориентированной (ОО) парадигмы. В ОО языках со статической типизацией ключевым решением стало совместное использование механизмов наследования и виртуализации. Наследование обеспечило идентичность интерфейсов родительского и дочерних классов, а

виртуализация позволила подменять методы в дочерних классах. В качестве примера можно привести языки C++ [7], Java [8] и другие.

Поддержку динамического полиморфизма, основанную на статической утиной типизации, стали включать и в процедурные языки. В языке Go [9], реализован механизм интерфейсов, позволяющий использовать в качестве обработчиков альтернатив функции, связанные со структурами данных. Близкий механизм на основе типажей реализован в языке программирования Rust [10]. Вместе с тем, представленные выше подходы напрямую не поддерживают эволюционного расширения программ в случае множественного полиморфизма, связанного, например, с реализацией мультиметодов. Помимо этого, даже в случае одиночного полиморфизма, определяемого также как монометод, изменение функциональности достаточно часто ведет к модификации интерфейсов и классов. Достижение необходимого эффекта, связанного с безболезненным расширением программ, возможно только за счет написания дополнительного кода, формирующего необходимые композиции и обертки над программными объектами, которые не связаны с вычислениями, определяемыми прикладной задачей. Подобные решения, в частности, предлагаются объектно-ориентированными паттернами проектирования [11].

Для инструментальной поддержки эволюционно расширяемого множественного полиморфизма была предложена процедурно–параметрическая (ПП) парадигма программирования [12], использующая альтернативные методы для повышения возможностей процедурного подхода. Используемые для ее реализации технические решения базируются на оригинальном параметрическом механизме формирования отношений между данными и обрабатывающими их процедурами, который может быть реализован различными способами [13]. Они обеспечивают безболезненное и независимое расширение как данных, так и функций в статически типизированных языках программирования. Для первоначальной апробации идеи был разработан язык O2M, расширяющий язык программирования Оберон-2 [14]. Проведенные на его основе эксперименты позволили определить возможности процедурно–параметрического программирования (ППП). Было показано, что подход может быть интегрирован как в уже существующие языки процедурного и функционального программирования, так и использоваться при разработке новых языков. Также показано, что ПП парадигма обеспечивает более гибкое расширение программ по сравнению с другими методами поддержки динамического полиморфизма для статически типизированных языков программирования [15].

В работе рассматривается интеграция ПП механизмов в язык программирования C [16], который входит в пятерку наиболее популярных языков по различным рейтингам. Язык обладает гибкостью и относительной простотой, широко используется в предметных областях, где ОО подход не является эффективным. Вместе с тем, многие решения, применяемые при разработке программ на языке C, могут приводить к ошибкам за счет отсутствия в ряде случаев контроля за типами данных во время компиляции. Такие ситуации возникают из-за того, что язык поддерживает произвольные преобразования типов, связанные с их разыменованием. Несмотря на то, что это зачастую позволяет достичь высокой производительности, подобный стиль кодирования снижает надежность программ. Другой проблемой является отсутствие в языке жесткой зависимости между альтернативными данными и идентифицирующими их признаками (например, в объединениях), которые могут произвольно вводиться и использоваться программистом. Это тоже приводит к ошибкам при разработке программ, выявляемым только во время выполнения.

Многие из этих проблем эффективно решаются в ОО языках программирования, а также в языках Go и Rust за счет использования полиморфизма. ПП парадигма, наряду с решением этих проблем [17], позволяет также эффективно поддерживать множественный полиморфизм и эволюционное расширение мультиметодов. Проведенное моделирование вносимых в язык изменений, реализованное на языке программирования C в рамках операционной системы Linux, подтверждает возможности такой реализации [18].

В результате проведенных расширений компилятора CLang [19] сформировано надмножество языка, названное процедурно-параметрическим C (procedural-parametric C, PPC) [20]. Это обеспечило проведение экспериментов по созданию процедурно-параметрических программ и позволило провести сопоставление методов гибкого программирования, предлагаемых данной парадигмой с методами и техникой кодирования, применяемых в других подходах.

ОО подход в настоящее время является доминирующим в различных предметных областях. Поэтому представляет интерес сравнение его возможностей с возможностями, представляемыми ПП парадигмой программирования. Примеры, демонстрирующие возможности ПП подхода и его сравнение с другими парадигмами, представлены в [21].

2. Особенности процедурно–параметрической парадигмы

При описании особенностей процедурно–параметрической парадигмы используются следующие понятия [18]:

- параметрическое обобщение – структура, объединяющая различные альтернативы и выступающая в качестве общего родительского типа для всех альтернативных подтипов;
- специализация параметрического обобщения – структура, добавляющая к обобщению одну из сформированных альтернатив, в которой основа специализации выступает в качестве подтипа;
- основа специализации – любая абстракция данных, допускаемая в качестве расширения альтернативы;
- экземпляр параметрического обобщения или специализированная переменная – переменная, имеющая тип обобщения и подтип одной из альтернативной специализации;
- обобщающая функция – функция, определяющая общий интерфейс для множества альтернативных специализаций;
- обработчик параметрической специализации или специализированная функция – функция, осуществляющая обработку одной из комбинаций специализаций и допускающая в качестве аргументов от одной и более специализаций, подставляемых вместо обобщающих аргументов;
- вызов параметрической функции – полиморфный вызов обобщающей функции с аргументами, являющимися специализациями.

Варианты специализаций, реализованные в языке PPC, приведены на рис. 1.

Параметрическое обобщение (или просто «обобщение») используется для объединения в единую категорию различных альтернатив. В отличие от типа `union` языка программирования C, заключающего альтернативы внутри описания, что не позволяет их расширять без изменения внутренней структуры, в PPC расширение обобщения альтернативами осуществляется децентрализовано. В первоначальном определении обобщение может содержать несколько альтернатив или не содержать ни одной альтернативы, определяя некоторую основу для последующего их подключения. Синтаксически обобщение определяется как структура, содержащая общие данные, которые также могут отсутствовать. Для задания альтернатив, используются угловые скобки. Например, описание пустой обобщенной фигуры может выглядеть следующим образом:

```
typedef struct Figure {}<> Figure;
```

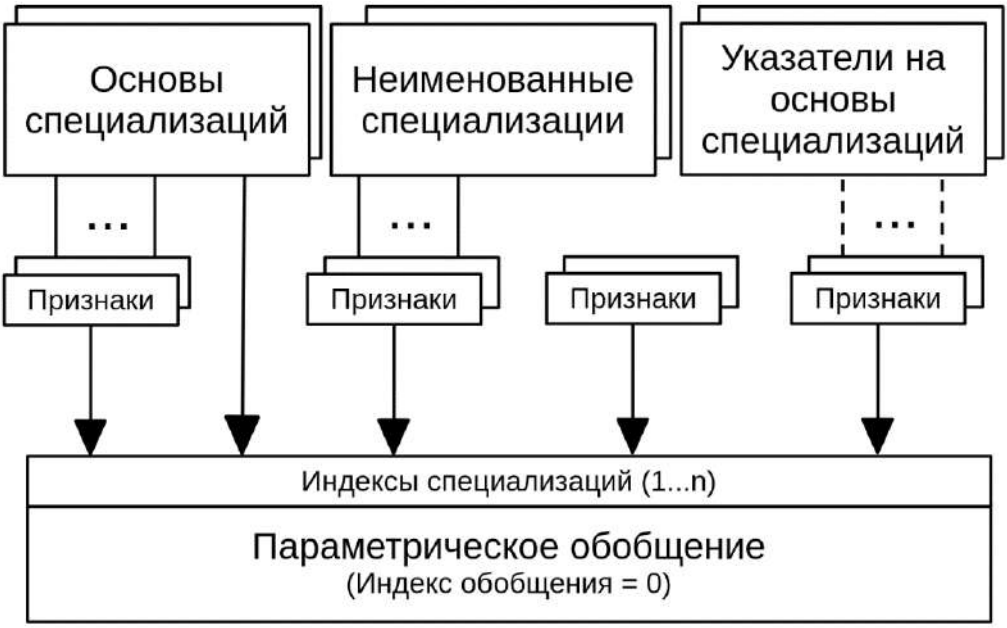


Рис. 1. Варианты формирования специализаций в языке программирования PPC.
Fig. 1. Options for the formation of specializations in the PPC programming language.

Использование описателя `typedef` не является обязательным. Однако в этом случае удобнее задавать описание типов структур только по именам. Обобщение может расширяться за счет добавления новых альтернатив, каждая из которых определяет одну из специализаций этого обобщения. В отличие от объединения языка C, расширение обобщения может осуществляться в различных единицах компиляции с использованием основ специализаций.

В качестве этих основ могут выступать именованные и неименованные типы данных, пустые типы и указатели на именованные типы данных. Именованные типы могут быть представлены базовыми типами и структурами. Например, в качестве такой структуры могут выступать явно задаваемые описания таких геометрических фигур, как прямоугольник и треугольник:

```
typedef struct Rectangle { int x, y; }    Rectangle;
typedef struct Triangle  { int a, b, c; } Triangle;
```

Добавление их в обобщение с формированием соответствующих специализаций будет выглядеть следующим образом:

```
Figure + < rect: Rectangle; >; // Создается фигура как прямоугольник
Figure + < Triangle; >;        // Создается фигура как треугольник
```

Формируемые при этом типы специализаций состоят из типа и подтипа. Тип определяется обобщением (`Figure`), а подтипы задаются либо явно признаками, либо непосредственно типом подключаемой основой специализации. Использование признаков обеспечивает уникальную идентификацию подтипа даже в том случае, когда одна и та же основа специализации многократно используется в качестве подтипа некоторого обобщения. Для приведенного обобщения в результате сформируются следующие типы специализаций:

```
Figure.rect        // Специализация фигуры - прямоугольника
Figure.Triangle    // Специализация фигуры - треугольника
```

Допускается формирование специализации с использованием неименованных структур. В этом случае для идентификации подтипа всегда используется признак:

```
Figure + < point: struct {int x, y;}; >; // Специализация фигуры - точки
```

Признак также используется при создании специализаций при отсутствии типа, что обеспечивается использованием ключевого слова `void`, а также в том случае, когда в качестве основы специализации выступает указатель на именованный тип. Например, таким образом можно описать набор специализаций, задающих дни недели. При этом в основной структуре можно указывать дополнительный параметр, определяющий номер недели:

```
// Дни недели
   struct WeekDay {int week_number;}
        <Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday: void;>
        const;
```

Ключевое слово `const` запрещает дальнейшее эволюционное расширение данного обобщения. При этом создаются специализации следующих типов:

```
WeekDay.Monday, WeekDay.Tuesday, WeekDay.Wednesday...
```

3. Сравнение объектно–ориентированного и процедурно–параметрического механизмов

Отличие подходов к реализации между объектно-ориентированным и процедурно–параметрическим полиморфизмом представлено на рис 2.

Объектно-ориентированный полиморфизм (рис. 2a) обеспечивается сочетанием наследования и виртуализации. Виртуализация при этом обычно реализуется за счет таблиц виртуальных методов (VT). Таблица базового класса Base, выступающего в роли обобщения, содержит указатели на один или несколько методов (F), которые обычно переопределяются в виртуальных таблицах производных классов, являющихся специализациями обобщения ($Child_1$–$Child_n$). При этом каждый производный класс, расширяя базовый класс, имеет собственный тип, что выражается в уникальном имени класса ($Child_i$). Производные классы формируются независимо друг от друга. Это позволяет эволюционно расширять альтернативные программные объекты, каждый из которых, при наличии одинаковых интерфейсов, может иметь иную функциональность, обеспечивая тем самым реализацию динамического ОО полиморфизма. Однако добавление нового виртуального метода требует модификации всей иерархии классов. Также данный механизм напрямую не поддерживает мультиметоды, для реализации которых ОО решением является использование диспетчеризации, не способствующей безболезненному расширению классов.

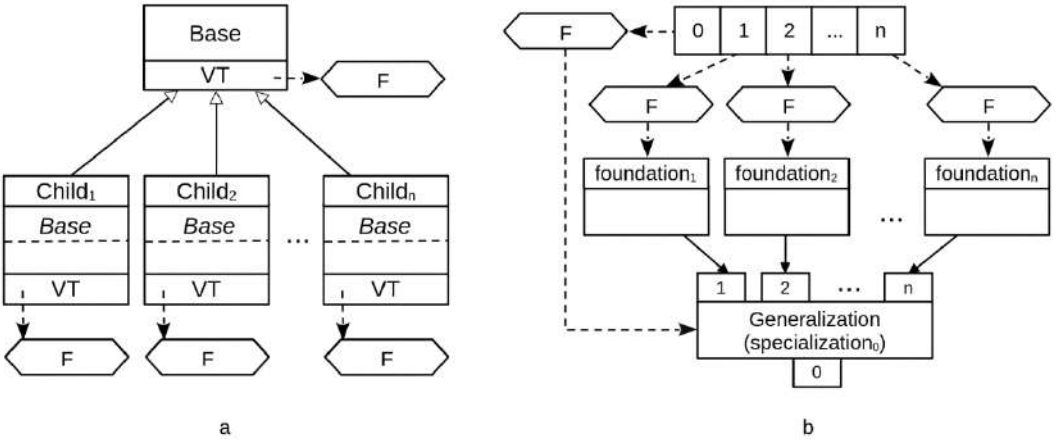


Рис. 2. Реализация ОО (a) и ПП (b) полиморфизма.
Fig. 2. Implementation of object-oriented (a) and procedural-parametric (b) polymorphis.

Процедурно–параметрический полиморфизм (рис. 2b) обеспечивает формирование альтернативных специализаций ($specialization_1$ – $specialization_n$), используя для этого основы специализаций ($foundation_1$ – $foundation_n$), которые являются подтипами данных. Добавление альтернативных специализаций к обобщающему их типу данных (Generalization) может осуществляться в произвольные моменты времени и в различных единицах компиляции. Формируемые при этом специализации обобщения принадлежат к тому же типу, что и обобщающий тип, являясь его подтипами. Само обобщение также трактуется как одна из специализаций ($specialization_0$), которая содержит только свои внутренние данные. Обработка сформированных альтернатив осуществляется с использованием внешних независимых специальных функций – обработчиков специализаций, расширяющих обобщающую их функцию. Они также могут добавляться независимо друг от друга. Механизм идентификации специализаций локализован внутри каждого обобщения с использованием внутренней индексации (параметрических индексов). Этот же механизм используется для автоматического выбора обработчиков специализаций через параметрические таблицы, каждая из которых связана со своим набором обобщающих функций. Добавление новых обобщающих функций и их обработчиков специализаций может осуществляться в любой момент времени независимо от уже существующих данных и функций, что характерно для процедурного подхода. Эти функции могут содержать произвольное число обобщающих аргументов, обеспечивая прямую реализацию эволюционно расширяемых мультиметодов [18].

3.1 Сопоставление базовых технических приемов

К базовым техническим приемам следует отнести формирование абстракций данных (классов) и функций (методов), обеспечивающих поддержку динамического полиморфизма. Их характеристики во многом определяют гибкую разработку программ и возможность их эволюционного расширения при добавлении новой функциональности. Непосредственное использование этих технических приемов позволяет не писать дополнительный код, напрямую не связанный с задачей предметной области.

Отличие между ОО и ПП подходами можно рассмотреть на простом примере обобщенной фигуры и таких ее специализаций как прямоугольник и треугольник. Пусть при использовании ОО парадигмы создается абстрактный базовый класс фигуры, содержащий чистый метод вычисления периметра. Помимо этого, пусть в данном классе присутствует общая переменная, задающая цвет фигуры и метод, обеспечивающий вывод этого параметра. Для демонстрации виртуального метода, не являющегося чистым, будем использовать вариант, проверяющий, что фигура является прямоугольником, который переопределяется в соответствующем производном классе.

```
class Figure {
  int color;
public:
  ...
  int GetColor() {return color;}
  virtual double Perimeter() = 0;
  virtual bool isRectangle() {return false;}
  ...
};
```

От этого класса формируются производные классы прямоугольника и треугольника.

```
class Rectangle: public Figure {
  int x, y;
public:
  ...
  virtual double Perimeter();
  virtual bool isRectangle() {return true;}
  ...
```

```
};
class Triangle: public Figure {
  int a, b, c;
public:
  ...
  virtual double Perimeter();
  ...
};
```

Реализация методов вычисления периметров может быть вынесена в независимые единицы компиляции.

```
double Rectangle::Perimeter() {
  return (double)(2*(x + y));
}
double Triangle::Perimeter() {
  return (double)(a + b + c);
}
```

При процедурно-параметрическом подходе в качестве альтернативы базовому классу можно создать обобщенную структуру, общее поле которой содержит цвет. Для запрета использования обобщения, не содержащего основы специализации, используется приравнивание к нулю.

```
typedef struct Figure { int color; }<> Figure = 0;
```

Аналогией подклассам является построение специализаций. Используем описанные выше структуры прямоугольника и треугольника, добавив их в обобщение с явными признаками.

```
Figure + < rect: Rectangle; >; // Фигура как прямоугольник
Figure + < trian: Triangle; >; // Фигура как треугольник
```

Вместо виртуальных методов формируются обработчики специализаций, которые являются расширениями обобщающей функции. Обобщенные аргументы этой функции переопределяются на специализации. За счет этого реализуется процедурно-параметрический полиморфизм [18], который для функции от одного полиморфного аргумента практически эквивалентен объектно-ориентированному полиморфизму. Эквивалентом чистого метода является абстрактная функция, которая должна быть обязательно переопределена на соответствующий обработчик для каждой специализации. У абстрактной обобщающей функции тело отсутствует.

```
double FigurePerimeter<Figure *f>() = 0; // Абстрактная обобщающая функция
// Обработчики специализаций
double FigurePerimeter<Figure.rect *f>() {
  return (double)(2*(f->@x + f->@y));
}
double FigurePerimeter<Figure.trian *f>() {
  return (double)(f->@a + f->@b + f->@c);
}
```

Для идентификации того, что осуществляется обращение к данным, предоставляемым основами специализаций, после ссылки «->» указывается признак основы «@». Возможно также использование обобщающих функций, эквивалентных виртуальным методам базового класса, содержащим реализацию. В этом случае аналогичная обобщающая функция ведет себя как обработчик по умолчанию для тех специализаций, которые не имеют собственного обработчика. Описанный выше виртуальный метод базового класса, проверяющий, является ли фигура прямоугольником, при ПП подходе будет реализован следующим образом: обобщающая функция будет выполнять проверку на прямоугольник для всех фигур, исключая сам прямоугольник.

```
_Bool isRectangle<Figure *f>() {return 0;}
```

Для прямоугольника будет сформирован соответствующий обработчик специализации.

```
_Bool isRectangle<Figure.rect *f>() {return 1;}
```

Как и в случае ОО подхода в этом случае не нужно реализовывать все обработчики специализаций. Достаточно реализовать только те из них, для которых требуется изменить функциональность.

Добавление новых функций для специализаций может осуществляться с передачей параметров через обычный список аргументов. Через этот же список специализации могут передаваться в функции, реализующие действия для переменных, расположенных внутри обобщений. Например, для вывода цвета фигуры может быть реализована обычная функция, получающая в качестве аргумента указатель на обобщение:

```
int GetColor(Figure *f) {return f->color;}
```

Так как специализации являются подтипами обобщения, они могут передаваться в эту функцию в качестве параметров.

Рассмотренные ситуации показывают, что процедурно-параметрический подход поддерживает основные методы формирования полиморфных отношений, аналогичные тем, что реализуются в ОО языках программирования со статической типизацией. Реализация полиморфизма в качестве расширения языка программирования С позволяет создавать более гибкий и надежный код за счет того, что вместо разыменования типов и прямого анализа вариантов при обработке альтернатив можно использовать обертки из обобщений и специализаций, обрабатываемые через вызов обобщающих функций [17]. Разбиение программы на отдельные функции, не содержащие общего анализа альтернатив, обеспечивают доступность кода для автоматического анализа, аналогичного по сложности анализу для языков С и С++.

Помимо этого, процедурно-параметрический полиморфизм обеспечивает более гибкое расширение кода по сравнению с ОО полиморфизмом. Это связано с использованием раздельного расширения данных и полиморфных функций. В случае ОО подхода добавление нового виртуального метода в базовый класс может привести к модификации наследуемых от него подклассов, особенно в том случае, если добавляется абстрактный метод.

Так как функции отделены от данных, то вопросы, как и в языке С, связанные с построением интерфейсов и использованием для их подключения множественного наследования, теряют смысл. Для группирования функций и их ограниченного использования соответствующими единицами компиляции достаточно сформировать соответствующие заголовочные файлы, содержащие прототипы необходимых обобщающих функций.

3.2 Мультиметоды и диспетчеризация

При процедурно-параметрическом подходе, мультиметоды могут быть непосредственно описаны в коде. Для этого в обобщающей функции достаточно указать несколько обобщенных аргументов [18]. Однако в ряде случаев, как и в статически типизированных ОО языках, для их реализации можно использовать диспетчеризацию. Такой прием может оказаться полезным, когда количество полиморфных аргументов в мультиметоде достаточно велико, а вариантов обработки комбинаций аргументов немного. Также при большом числе аргументов можно сочетать диспетчеризацию и мультиметоды для различных комбинаций аргументов.

В качестве примера, демонстрирующего возможности реализации диспетчеризации, можно рассмотреть мультиметод, осуществляющий анализ вложенности первой геометрической фигуры во вторую. Ранее данный пример рассматривался для демонстрации возможностей эволюционного расширения мультиметодов при процедурно-параметрическом подходе [18]. При первоначальном наличии только двух альтернативных специализаций (прямоугольника

и треугольника) вместо непосредственного описания обобщающей функции, определяющей мультиметод, формируются две обобщающие функции от одного полиморфного аргумента:

```
// Обобщающая функция, задающая вход в первую фигуру.
// Вторая фигура передается как обычный параметр
void Multimethod<Figure* f1>(Figure* f2) {} = 0;
```

Обработчики конкретных специализаций определяют полиморфный вход во вторую фигуру, относительно известной им специализации первого аргумента, запуская обобщенные функции, выполняющие обработку в зависимости от передаваемых в них специализаций:

```
// Обработчик специализации, когда первая фигура - прямоугольник
void Multimethod<Figure.rect* r1>(Figure* f2) {
  MultimethodFirstRect<f2>(r1);
}

// Обработчик специализации, когда первая фигура - треугольник
void Multimethod<Figure.trian* t1>(Figure* f2) {
  MultimethodFirstTrian<f2>(t1);
}
```

Обобщенные вспомогательные функции для обработки второго аргумента получают уже известную специализацию определяющие первый аргумент:

```
// Обобщающая функция, задающая вход во вторую фигуру,
// Когда первая фигура уже определена и это прямоугольник
static void MultimethodFirstRect<Figure* f2>(Figure.rect* r1) = 0;

// Обобщающая функция, задающая вход во вторую фигуру,
// Когда первая фигура уже определена и это треугольник
static void MultimethodFirstTrian<Figure* f2>(Figure.trian* t1) = 0;
```

Для каждой из четырех возможных комбинаций создается свой обработчик специализации для второго аргумента:

```
// Обработчик специализации для двух прямоугольников
static void MultimethodFirstRect<Figure.rect* r2>(Figure.rect* r1) {
  printf("Rectangle - Rectangle Combination\n");
}
// Обработчик специализации для прямоугольника и треугольника
static void MultimethodFirstRect<Figure.trian* t2>(Figure.rect* r1) {
  printf("Rectangle - Triangle Combination\n");
}
// Обработчик специализации для треугольника и прямоугольника
static void MultimethodFirstTrian<Figure.rect* r2>(Figure.trian* t1) {
  printf("Triangle - Rectangle Combination\n");
}
// Обработчик специализации для двух треугольников
static void MultimethodFirstTrian<Figure.trian* t2>(Figure.trian* t1) {
  printf("Triangle - Triangle Combination\n");
}
```

В целом можно отметить, что диспетчеризация внешне похожа на ОО решение. Однако при добавлении нового подкласса, например, для круга, в ОО подходе приходится изменять ранее написанные родительский класс и его подклассы. При процедурно-параметрическом программировании новые альтернативы добавляются без изменения ранее написанного кода. К еще одной положительной черте ПП подхода можно отнести то, что передаваемый пользователю интерфейс мультиметода, реализованного через диспетчеризацию, можно ограничить только внешней обобщающей функцией:

```
// Сигнатура для входа в диспетчеризацию мультиметода
void Multimethod<Figure* f1>(Figure* f2);
```

Промежуточные функции могут быть скрыты в отдельных единицах компиляции. Возможно как их отсутствие в заголовочных файлах, так и использование дополнительного ключевого слова `static`. В случае ОО подхода все промежуточные методы засоряют интерфейсы базового и производных классов, не используясь при этом во внешних взаимодействиях.

3.3 ПП реализация паттернов ОО проектирования

Для сопоставления с возможностями ОО программирования проведена реализация классических паттернов ОО проектирования, представленных в работе [11]. Практически для всех паттернов была реализована аналогичная функциональность и обеспечены соответствующие критерии качества. Независимость данных и функций в языке С в большинстве случаев позволили избавиться от ряда классов, ориентированных только на поддержку полиморфных интерфейсов, заменив их на эволюционно расширяемый перечислимый тип. Это же позволило обеспечить более гибкое расширение как полиморфных функций, так и данных без изменения ранее написанного кода практически для всех паттернов, в которых при ОО подходе присутствуют соответствующие ограничения. В ряде случаев появилась возможность использования более простых и эффективных решений, обуславливаемых как с особенностями традиционного процедурного подхода, так спецификой, добавляемой процедурно–параметрическим механизмом.

В качестве примера, демонстрирующего особенности использования процедурно-параметрического подхода, можно рассмотреть паттерн Visitor. Полностью его ОО реализация на языке C++ приведена в [22]. Упрощенно данный паттерн может быть представлен следующим кодом:

```
class ConcreteComponentA;
class ConcreteComponentB;
class Visitor { // Абстрактный базовый класс для посетителя
 public:
  // Требует изменения интерфейса при добавлении новых конкретных компонент
  virtual void VisitConcreteComponentA
       (const ConcreteComponentA *element) const = 0;
  virtual void VisitConcreteComponentB
       (const ConcreteComponentB *element) const = 0;
};
class Component {
 public:
  virtual ~Component() {}
  virtual void Accept(Visitor *visitor) const = 0;
};
class ConcreteComponentA : public Component {
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentA(this);
  }
  void ExclusiveMethodOfConcreteComponentA() const {...}
};
class ConcreteComponentB : public Component {
 public:
  void Accept(Visitor *visitor) const override {
    visitor->VisitConcreteComponentB(this);
  }
  void SpecialMethodOfConcreteComponentB() const {...}
};
class ConcreteVisitor1 : public Visitor {
 public:
  void VisitConcreteComponentA
       (const ConcreteComponentA *element) const override {
    element->ExclusiveMethodOfConcreteComponentA();
    // ... Специфические манипуляции A в ConcreteVisitor1
  }
```

```
    void VisitConcreteComponentB
        (const ConcreteComponentB *element) const override {
      element->SpecialMethodOfConcreteComponentB();
      // ... Специфические манипуляции B в ConcreteVisitor1
    }
};

class ConcreteVisitor2 : public Visitor {
 public:
    void VisitConcreteComponentA
        (const ConcreteComponentA *element) const override {
      element->ExclusiveMethodOfConcreteComponentA();
      // ... Специфические манипуляции A в ConcreteVisitor2
    }
    void VisitConcreteComponentB
        (const ConcreteComponentB *element) const override {
      element->SpecialMethodOfConcreteComponentB();
      // ... Специфические манипуляции B в ConcreteVisitor2
    }
};
```

Базовый класс посетителя является интерфейсом и переопределен в подклассах. Очевидно, что добавление нового конкретного компонента ведет к изменению этого интерфейса и его подклассов. Также можно отметить использование диспетчеризации.

Прямая процедурно-параметрическая имитация этого паттерна, также может быть выполнена с использованием диспетчеризации:

```
// Конкретные компоненты могут содержать любые данные
typedef struct ConcreteComponentA {} ConcreteComponentA;
// Функция, выполняемая конкретным компонентом A
const char* ExclusiveMethodOfConcreteComponentA(ConcreteComponentA* c) {
  return "A";
}

typedef struct ConcreteComponentB {} ConcreteComponentB;
// Функция, выполняемая конкретным компонентом B
const char* SpecialMethodOfConcreteComponentB(ConcreteComponentB* c) {
  return "B";
}

// Посетитель как обобщение, имитирующее перечислимый тип
typedef struct Visitor {}<> Visitor;
// Обобщающие функции, реализуемые для различных компонентов
void VisitConcreteComponentA<Visitor* v>(const ConcreteComponentA *c) = 0;
void VisitConcreteComponentB<Visitor* v>(const ConcreteComponentB *c) = 0;

// Компоненты также могут быть подтипами обобщения
typedef struct Component {}<> Component;
// Обобщающая функци доступа к компоненту с передачей посетителя
void Accept<Component* c>(Visitor *v) = 0;

// Каждый специализированный Компонент должен реализовать Accept,
// чтобы он вызывал метод посетителя, соответствующий типу компонента.
Component + <ConcreteComponentA;>;
// Доступ к компоненту A
void Accept<Component.ConcreteComponentA* c>(Visitor *v) {
  VisitConcreteComponentA<v>(&(c->@));
}
Component + <ConcreteComponentB;>;
// Доступ к компоненту B
void Accept<Component.ConcreteComponentB* c>(Visitor *v) {
  VisitConcreteComponentB<v>(&(c->@));
}
```

```
// Создание конкретных посетителей и обработчиков ими конкретных компонент.
Visitor + <ConcreteVisitor1: void;>;
void VisitConcreteComponentA
     <Visitor.ConcreteVisitor1* v>(ConcreteComponentA *c) {
  const char* str = ExclusiveMethodOfConcreteComponentA(c);
  printf("%s + ConcreteVisitor1\n", str);
}
void VisitConcreteComponentB
     <Visitor.ConcreteVisitor1* v>(ConcreteComponentB *c) {
  char* str = SpecialMethodOfConcreteComponentB(c);
  printf("%s + ConcreteVisitor1\n", str);
}
Visitor + <ConcreteVisitor2: void;>;
void VisitConcreteComponentA
     <Visitor.ConcreteVisitor2* v>(ConcreteComponentA *c) {
  char* str = ExclusiveMethodOfConcreteComponentA(c);
  printf("%s + ConcreteVisitor2\n", str);
}
void VisitConcreteComponentB
     <Visitor.ConcreteVisitor2* v>(ConcreteComponentB *c) {
  char* str = SpecialMethodOfConcreteComponentB(c);
  printf("%s + ConcreteVisitor2\n", str);
}
```

Однако, вместо абстрактного интерфейса используется эволюционно расширяемый перечислимый тип, сформированный на основе обобщения и обеспечивающий гибкое расширение в комбинации с обобщающей функцией и обработчиками специализаций для каждого конкретного компонента. При этом добавление новых данных позволяет безболезненно расширять Посетителя без изменения ранее написанного кода.

Диспетчеризацию также можно заменить на мультиметод, что позволяет получить более простое решение:

```
typedef struct ConcreteComponentA {} ConcreteComponentA;
// Функция, выполняемая конкретным компонентом A
const char* ExclusiveMethodOfConcreteComponentA(ConcreteComponentA* c) {
  return "A";
}
typedef struct ConcreteComponentB {} ConcreteComponentB;
// Функция, выполняемая конкретным компонентом B
const char* SpecialMethodOfConcreteComponentB(ConcreteComponentB* c) {
  return "B";
}
typedef struct Component {}<> Component;
Component + <ConcreteComponentA;>;
Component + <ConcreteComponentB;>;
typedef struct Visitor {}<> Visitor;
Visitor + <ConcreteVisitor1: void;>;
Visitor + <ConcreteVisitor2: void;>;
// Посетитель и компонент образуют мультиметод
void VisitComponent<Visitor* v, Component *c>() = 0;
// Обработчики специализаций, реализованные через мультиметод
void VisitComponent
     <Visitor.ConcreteVisitor1* v, Component.ConcreteComponentA *c>() {
  const char* str = ExclusiveMethodOfConcreteComponentA(&(c->@));
  printf("%s + ConcreteVisitor1\n", str);
}
void VisitComponent
     <Visitor.ConcreteVisitor1* v, Component.ConcreteComponentB *c>() {
  const char* str = SpecialMethodOfConcreteComponentB(&(c->@));
  printf("%s + ConcreteVisitor1\n", str);
}
void VisitComponent
     <Visitor.ConcreteVisitor2* v, Component.ConcreteComponentA *c>() {
  const char* str = ExclusiveMethodOfConcreteComponentA(&(c->@));
```

```
    printf("%s + ConcreteVisitor2\n", str);
}
void VisitComponent
     <Visitor.ConcreteVisitor2* v, Component.ConcreteComponentB *c>() {
  const char* str = SpecialMethodOfConcreteComponentB(&(c->@));
  printf("%s + ConcreteVisitor2\n", str);
}
```

Представленное решение по сути является прямой реализацией вариантов обработки нужных компонент, позволяя быстро и безболезненно добавлять как новые компоненты, так и варианты их обхода без изменения ранее написанного кода.

4. Заключение

Проведенное сравнение показывает, что процедурно-параметрическая парадигма программирования обеспечивает более гибкую разработку программ по сравнению с ОО подходом. При этом реализация соответствующей надстройки над языком программирования С позволяет разрабатывать более надежные и эволюционно расширяемые программы даже в рамках этого языка. Помимо этого, обеспечивается добавление новых альтернативных данных, а также функций, позволяющих безболезненно для ранее написанного кода использовать множественный полиморфизм.

Список литературы / References

[1]. Фуксман, А. Л. Технологические аспекты создания программных систем. / А. Л. Фуксман – М.: Статистика, 1979. – 184 с.

[2]. Горбунов-Посадов, М. М. Система открыта, но что-то мешает. / М. М. Горбунов-Посадов // Открытые системы. 1996. № 6. С. 36–39.

[3]. Горбунов-Посадов, М. М. Конфигурационные ориентиры на пути к многократному использованию. / М. М. Горбунов-Посадов – ИПМ им. М.В.Келдыша РАН. Препринт № 37, 1997 г.

[4]. Горбунов-Посадов, М. М. Облик многократно используемого компонента. / М. М. Горбунов-Посадов // Открытые системы. 1998. № 3. С. 45–49.

[5]. Горбунов-Посадов, М. М. Расширяемые программы. / М. М. Горбунов-Посадов – М.: Полиптих, 1999.

[6]. Горбунов-Посадов, М. М. Эволюция программы: структура транзакции. / М. М. Горбунов-Посадов // Открытые системы. 2000, № 10. С. 43–47.

[7]. M. Gregoire, Professional C++, John Wiley & Sons. 2018, p. 1122.

[8]. E. Sciore, Java Program Design, Apress Media. 2019, p. 1122.

[9]. A. Freeman, Pro Go: The Complete Guide to Programming Reliable and Efficient Software Using Golang, Apress. 2022, p. 1105.

[10]. J. Blandy, J. Orendorff, and L. F. Tindall, Programming Rust, O'Reilly Media. 2021, p. 735.

[11]. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Professional. 1994, p. 416.

[12]. Легалов А.И. Процедурно–параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с. Доступна в сети Интернет: http://www.softcraft.ru/ppp/pppfirst/, accessed 30.08.2024.

[13]. Легалов И.А. Применение обобщенных записей в процедурно–параметрическом языке программирования // Научный вестник НГТУ, 2007. № 3 (28). С. 25–38.

[14]. Легалов А.И. Швец Д.А. Процедурный язык с поддержкой эволюционного проектирования. --- Научный вестник НГТУ, № 2 (15), 2003. С. 25-38.

[15]. Легалов А.И., Косов П.В. Эволюционное расширение программ с использованием процедурно–параметрического подхода // Вычислительные технологии. 2016. Т. 21. № 3. С. 56–69.

[16]. Прата С. Язык программирования С. Лекции и упражнения, 5-е издание. : Пер. с англ. – М.: Издательский дом «Вильямс», 2013. – 960 с.

[17]. Легалов А.И., Косов П.В. Процедурно-параметрический полиморфизм в языке с для повышения надежности программ. // XIV Всероссийское совещание по проблемам управления, ВСПУ-2024,

Москва 17-20 июня 2024 г. С. 2827-2833. https://vspu2024.ipu.ru/preprints/2827.pdf, accessed 30.08.2024.

[18]. Легалов А.И., Косов П.В. Расширение языка С для поддержки процедурно–параметрического полиморфизма. Моделирование и анализ информационных систем. 2023;30(1):40-62. doi: 10.18255/1818-1015-2023-1-40-62.

[19]. Clang: A c language family frontend for llvm. [Online]. Available: https://clang.llvm.org/, accessed 05.06.2025.

[20]. Язык процедурно-параметрического программирования РРС. Репозиторий компилятора языка на Gitverse: https://gitverse.ru/kpdev/llvm-project. Ветка pp-extension-v2, accessed 05.06.2025.

[21]. Примеры на Гитхаб, написанные с использованием процедурно–параметрической версии языка С: https://github.com/kreofil/evo-situations, accessed 05.06.2025.

[22]. Реализация паттерна ОО проектирования Visitor на языке программирования С++: https://refactoringguru.cn/ru/design-patterns/visitor/cpp/example, accessed 02.09.2025.

Информация об авторах / Information about authors

Павел Владимирович КОСОВ – аспирант факультета компьютерных наук НИУ «Высшая школа экономики». Его научные интересы связаны с разработкой языков программирования и компиляторов, оптимизацией LLVM, эволюционной разработкой программного обеспечения.

Pavel Vladimirovich KOSOV is a postgraduate student of the Faculty of Computer Science at the Higher School of Economics. His research interests are related to the development of programming languages and compilers, LLVM optimization, and evolutionary software development.

Александр Иванович ЛЕГАЛОВ – доктор технических наук, профессор факультета компьютерных наук НИУ «Высшая школа экономики». Его научные интересы: технологии программирования, эволюционная разработка программного обеспечения, архитектурно-независимое параллельное программирование, языки программирования.

Alexander Ivanovich LEGALOV – Dr. Sci. (Tech.), Professor of the Faculty of Computer Science at the Higher School of Economics. His research interests include software engineering, evolutionary software development, architecture-independent parallel programming, programming languages.

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

Статический анализ исходного кода для языка Golang: обзор литературы

Дворцова В. В., ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>
Бородин А.Е., ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Методы статического анализа определяют свойства программы без ее выполнения, при этом различные свойства позволяют решать различные задачи. Мы выполнили обзор статей, посвященных статическому анализу Golang. В данной работе мы изучили 34 публикации, опубликованные с момента выхода первой версии языка Go 1.0 (с 2012 по 2025 год включительно), посвящённые статическому анализу исходного кода на языке Golang. На основе проведённого анализа мы выделили основные направления и методы использования статического анализа, а также рассмотрели используемые промежуточные представления, особенности языка Golang, влияющие на процесс анализа, и трудности, с которыми сталкиваются разработчики статических анализаторов. Этот обзор будет полезен как разработчикам статических анализаторов, так и разработчикам программ на языке Golang, предоставляя им систематизированное понимание текущего состояния исследований в области статического анализа исходного кода на языке Golang.

Ключевые слова: язык программирования Golang; статический анализ; обзор литературы.

Для цитирования: Дворцова В. В., Бородин А.Е. Статический анализ исходного кода для языка Golang: обзор литературы. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 59–82. DOI: 10.15514/ISPRAS–2025–37(6)–4.

Static Analysis of Golang Source Code: A Survey

V.V. Dvortsova, ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>
A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>

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

Abstract. Static analysis methods determine the properties of a program without executing it, while different properties allow solving different tasks. We have reviewed articles on Golang static analysis. In this paper, we have reviewed 34 papers published since the release of Go 1.0 (2012 – 2025) and focused on static analysis in Golang. Based on our analysis, we have identified the main trends and methods for performing static analysis as well as intermediate representations and features of Golang that affect the process. We have also examined the challenges faced by developers of static analyzers. This survey will be helpful for both developers of static analyzers and Golang developers, providing a systematic understanding of current research in static analysis for Go.

Keywords: Static analysis, Golang, Survey.

For citation: Dvortsova V.V., Borodin A.E. Static Analysis of Golang Source Code: A Survey. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 59-82 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-4.

1. Введение

С момента своего релиза 28 марта 2012 года язык программирования Golang (далее Go) [1] получил широкое применение среди разработчиков программного обеспечения. Go – язык с открытым исходным кодом, в нем сочетается нативная компиляция, статическая типизация, автоматическое управление памятью и упрощенный синтаксис. Как и любой другой язык программирования, Go не защищен от ошибок, которые могут возникнуть на этапах проектирования, написания или поддержки кода. Эти ошибки могут привести к уязвимостям безопасности или снижению производительности. Одним из ключевых подходов к минимизации количества ошибок и улучшению качества кода является статический анализ исходного кода. Статический анализ позволяет выявлять потенциальные проблемы на ранних этапах разработки без запуска программы, что значительно снижает затраты на исправление ошибок и повышает качество программного обеспечения.

Разработчики языка встроили в него легковесный статический анализатор `Go vet` [2], который ищет подозрительные шаблоны на абстрактном синтаксическом дереве (АСД представляет собой структурированное представление исходного кода в виде дерева, где каждый узел соответствует определенной конструкции языка программирования (например, операторы, выражения, функции), такие как несоответствия строки и аргументов формата в методе `fmt.Printf`, бесполезные сравнения между функциями и `nil`, неиспользуемые результаты вызовов некоторых функций и другие. Также создатели языка разработали фреймворк статического анализа `golang.org/x/tools` [3], с помощью которого можно создавать разнообразные анализаторы для широкого круга задач.

Несмотря на растущий интерес к статическому анализу Go, исследования в этой области остаются несистематизированными. Существующие работы охватывают широкий спектр задач, включая анализ параллелизма, обнаружение уязвимостей в зависимостях, оптимизацию управления памятью и выявление «запахов кода». Однако отсутствует целостный обзор, который бы систематизировал текущие достижения, методы и ограничения статического анализа для Go.

Статья структурирована следующим образом:

- в разделе 2 представлен обзор инструментов и характеристик, связанных со статическим анализом кода на языке Go: в подразделе 2.1 описываются ключевые

особенности языка Go, оказывающие влияние на процесс статического анализа, в подразделе 2.2 рассматриваются популярные анализаторы исходного кода;

- в разделе 3 рассматриваются существующие работы, посвященные обзорам исследований в области языка Go;
- в разделе 4 подробно описывается методология проведения обзора, включая используемые источники данных и формулировку исследовательских вопросов;
- в разделе 5 представлен систематизированный обзор исследований и их особенностей реализации: в подразделе 5.1 дается краткий анализ найденных публикаций, классифицированных по основным направлениям исследований, в подразделе 5.2 рассматриваются ключевые аспекты реализации анализа, включая промежуточное представление, используемые методы анализа и ограничения соответствующих работ;
- в разделе 6 обсуждаются ограничения данного исследования;
- в разделе 7 подводятся итоги работы и формулируются выводы.

2. Анализ и характеристики языка Go

2.1 Особенности языка Go

Дизайн языка Golang [1, 4-7] напрямую влияет на подходы к статическому анализу. Ниже перечислены ключевые особенности языка, которые формируют специфику анализа кода:

- Строгая типизация и компиляция:
  - Строгая типизация уменьшает количество ошибок времени выполнения, но увеличивает важность анализа типов на этапе компиляции.
  - Компилятор Go уже выполняет базовый статический анализ, выявляя очевидные ошибки, такие как использование неинициализированных переменных или неверное приведение типов, недостижимый код. Однако для более глубокого анализа требуются сторонние инструменты.
- Параллелизм:
  - Горутины и каналы являются источниками сложных ошибок, например, гонок данных и взаимоблокировок, а также усложняют поток управления программы. Статический анализ для выявления этих проблем требует применения специализированных методов, таких как анализ потока данных и моделирование взаимодействия между потоками.
- Управление ресурсами:
  - В язык встроен сборщик мусора, тем не менее, разработчики часто сталкиваются с утечками ресурсов, например, незакрытыми файловыми дескрипторами или соединениями с базами данных.
- Конструкции, усложняющие поток управления:
  - Инструкция `defer` отложенного вызова функции;
  - Замыкания.
- Другие особенности:
  - Встраивание типов (Struct Embedding);
  - Дженерики (Generics);
  - Коллекции в Go (`map`, `slice`);

- Полиморфизм через интерфейсы;
- Встроенная поддержка нескольких возвращаемых значений;
- Отсутствие исключений и механизма обработки исключений (например, как в Java, Kotlin, C++);
- Отсутствие классов и механизма наследования (например, как в языках Java, Kotlin, C++);
- Отсутствие перегрузки функций (Function overloading).

Вышеперечисленные особенности языка влияют на то, с какими проблемами и уязвимостями можно встретиться в исходном коде.

Статья [6] посвящена анализу ошибок, возникающих в экосистеме языка Go, с целью их понимания и классификации. Авторы статьи проанализировали 51020 отчетов об ошибках (issue reports) в репозитории Go на GitHub. Анализ ошибок помогает выявить слабые места языка. Среди ключевых причин, приводящих к ошибкам, авторы выделили следующие (от частых к редким):

1. неправильная логика кода;
2. некорректная проверка условия;
3. некорректная инициализация переменных;
4. некорректный вызов функции;
5. неправильная обработка исключительных ситуаций и предупреждений;
6. ошибки, связанные с параллелизмом;
7. ошибки при работе с памятью.

Исследование предоставляет данные для создания инструментов анализа, которые могут автоматически обнаруживать и предотвращать распространенные ошибки.

Статья [7] посвящена обзору возможностей линтеров (легковесных статических анализаторов, ищущих шаблоны кода, приводящие к различным ошибкам). В ней выделены основные проблемы исходного кода 20-ти открытых проектов, с которыми линтеры должны справляться (из отчетов по проектам на GitHub), и сформулировали 10 категорий ошибок:

1. слабая ссылка на элемент слайса внутри цикла (Weak element reference in slices) (неактуально после версии Go 1.22);
2. пропуск обработки ошибок (Missing error handling statement);
3. потенциально ошибочные использования операций сравнения (Loose boolean expressions);
4. неправильное использование горутин (Misuse of goroutine);
5. неиспользуемые параметры (Unused parameters);
6. избыточные и неполные объявления типов (Misuse of type conversion and declaration);
7. отсутствие defer (Missing defer);
8. отсутствие проверки на nil (Missing a null check);
9. неправильная обработка специальных символов при кодировании/декодировании (Failed to remove special characters in decoding and encoding);
10. отсутствие обработок ошибок (Absent error handling).

В работе [7] также была исследована работа пяти популярных линтеров: `errcheck` [8], `Gosec` [9], `Go vet`, `revive` [10], `staticcheck` [11]. Авторы отмечают низкую производительность и низкую эффективность этих инструментов, а также отсутствие линтера, который бы умел находить все виды ошибок из указанных категорий (в среднем

линтер мог находить ошибки из одной или двух категорий). Делается вывод, что для повышения качества кода важно использовать комбинацию различных инструментов.

2.2 Статический анализ Go

Для Go разработано множество статических анализаторов (например, Go `vet`, `Gosec`, `errcheck`, `staticcheck`), ищущих различные ошибки в исходном коде, некоторые из них агрегируют в себе другие анализаторы (`golangci-lint` [12]). Так, результаты опроса разработчиков Go 2024 H2 [13] показали, что самым популярным инструментом анализа кода является `gopls` [14], который использовали 65% респондентов, поскольку `gopls` – языковый сервер, разработанный создателями языка Go и используемый в редакторе кода VS Code [15], популярном среди разработчиков на Go. Реже использовались `golangci-lint` (57% респондентов) и `staticcheck` (34%). Только 10% респондентов указали, что не используют никаких инструментов анализа.

В исследовании [16] описывается влияние особенностей языка программирования (в том числе и языка Go) на качество написанных на нем программ. Исследователи проанализировали большое количество открытых проектов на GitHub. Авторы отметили, что есть хоть и небольшая, но существенная связь между особенностями языка и возможными дефектами в программах на нем: функциональные, строго типизированные языки вызывали меньше ошибок, чем процедурные, слабо типизированные, без автоматического управления памятью. Также была отмечена зависимость некоторых типов дефектов, например, ошибок параллелизма и утечек памяти, от имеющихся в языке примитивов. Go имеет строгую статическую типизацию и автоматическое управление памятью, поэтому в программах на нем меньше потенциальных ошибок, чем в программах на языках с динамической типизацией (например, Python, JavaScript) и отсутствием сборщика мусора (например, C/C++).

Отметим, что задач, которые решаются методами статического анализа, много. Так, например, в статье [17] описываются существующие автоматические инструменты статического анализа для Go, их возможности и сценарии использования. Среди них: инструменты генерации кода (`Mockgen`, `Mockery`, `Scaffold`, `Gqlgen`, `gRPC`), инструменты для поиска ошибок или нарушения свойств программы (`Check` [18], `Dupl` [19], `Errcheck` [8], `Gocyclo` [20], `Gosec` [9], `Prealloc` [21], `Safesql` [22]), форматтеры кода (`gofmt` [23]), отладчик (`Delve` [24]) и генератор документации (`Godoc` [25]).

3. Связанные работы

Насколько нам известно, для языка Go отсутствует целостный обзор литературы, который бы охватывал различные методы, инструменты и ограничения статического анализа в контексте Go. Однако в литературе было предложено несколько обзоров, связанных с безопасностью приложений на Go. В уже упомянутой статье [7] ее авторы сделали обзор возможностей пяти популярных линтеров на 20 открытых проектах. В также уже упомянутой статье [17] описываются возможности и сценарии использования существующих автоматических инструментов статического и динамического анализа для языка Go.

4. Методология обзора литературы

Методология, которой мы следовали при создании обзора, основана на рекомендациях Kitchenham [26]. Эта методология уже использовалась другими обзорами [27-28].

Мы изучили 34 статьи, связанных со статическим анализом Go (полный список приведен в табл. 1).

Табл. 1. Полный список рассмотренных публикаций.
Table 1. The complete list of surveyed papers.

Год	Статья
2016	[29] Information flow analysis for Go
2016	[30] Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis
2016	[31] Detection of Bugs and Code Smells through Static Analysis of Go Source Code
2017	[32] Fencing off Go: liveness and safety for channel-based programming
2018	[33] A static verification framework for message passing in Go using behavioural types
2019	[34] An empirical study of messaging passing concurrency in Go projects
2019	[35] Verifying message-passing programs with dependent behavioural types.
2019	[36] Go-Sanitizer: Bug-Oriented Assertion Generation for Golang.
2019	[37] Godexpo: an automated god structure detection tool for golang.
2020	[38] Static race detection and mutex safety and liveness for Go programs
2020	[39] Escape from escape analysis of Golang
2020	[40] Uncovering the hidden dangers: Finding unsafe Go code in the wild
2021	[41] Breaking type safety in Go: an empirical study on the usage of the unsafe package
2021	[42] Interprocedural static analysis for finding bugs in Go programs
2021	[43] Static analyzer for Go
2021	[44] Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace
2021	[45] Automated verification of Go programs via bounded model checking
2021	[46] GoDetector: Detecting concurrent bug in Go
2021	[47] Analysing GoLang Projects' Architecture Using Code Metrics and Code Smell
2021	[48] Gobra: Modular specification and verification of go programs
2021	[49] Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems
2022	[50] How many mutex bugs can a simple analysis find in Go programs?
2022	[51] Cryptogo: Automatic detection of Go cryptographic api misuses
2022	[52] Detecting blocking errors in Go programs using localized abstract interpretation
2022	[53] Interprocedural static analysis for Go with closure support
2022	[54] Devirtualization for static analysis with low level intermediate representation
2023	[55] Static Analysis for Go: Build Interception
2024	[56] Статический анализ ассоциативных массивов в Go
2024	[57] GoGuard: Efficient Static Blocking Bug Detection for Go
2024	[58] Gopher: High-Precision and Deep-Dive Detection of Cryptographic API Misuse in the Go
2024	[59] Golang Defect Detection based on Value Flow Analysis
2024	[60] GoSurf: Identifying Software Supply Chain Attack Vectors in Go
2024	[61] MEA2: A Lightweight Field-Sensitive Escape Analysis with Points-to Calculation for Golang
2025	[62] An Empirical Study of CGO Usage in Go Projects–Distribution, Purposes, Patterns and Critical Issues.

При поиске статей использовались ключевые слова, такие как *static analysis*, *Golang*, *Go*, *Go programming language*, *code quality tools*, и их комбинации. Поиск проводился в научных базах данных (Google Scolar [63], IEEE Xplore [64], ACM Digital Library [65], SpringerLink [66]) и открытых источниках (GitHub [67]). В обзор включались статьи, описывающие инструменты или методы статического анализа, специфичные для Go, а также обзоры, использующие статический анализ для проведения исследования [34, 41, 62]. Исключались работы, не связанные с Go, а также работы, сосредоточенные только на динамическом анализе.

Данный обзор направлен на решение следующих исследовательских вопросов:

Вопрос 1: Каковы цели статического анализа?

В рамках данного вопроса мы рассмотрим использование статического анализа в контексте повышения качества кода и выделим его основные направления. Другие цели применения статического анализа, не связанные с качеством кода, в этой работе не рассматриваются.

Вопрос 2: Какие методы статического анализа используются?

В этом исследовательском вопросе мы подробно изучаем основные разработанные методы анализа. С этой целью мы исследуем следующие подвопросы:

2.1 Какие промежуточные представления используются при анализе?

2.2 Какие особенности, характерные для Go, принимаются во внимание?

2.3 С какими трудностями сталкиваются разработчики статического анализа приложений на Go?

5. Анализ литературы

После изучения указанных в табл. 1 статей мы выполнили их обзор, разбив их на темы, посвященные определенным ошибкам. Для каждой статьи из направления мы кратко описали основную цель статьи и используемые методы анализа, а также ограничения работы, если таковые были указаны в изучаемой публикации.

На рис. 1 показана динамика количества статей по годам в период с 2012 по 2025 год. Видно, что наибольшее количество статей приходится на 2021 год (9 публикаций), а минимальное – на 2017, 2018 и 2023 годы (по одной публикации).

5.1 Цели статического анализа

Мы выделили 10 направлений исследований (табл. 2 показывает разбиение статей по направлениям), среди которых:

- параллелизм в Go;
- неправильное использование криптографических библиотек;
- неправильное использование пакета unsafe;
- ошибки в CGO коде;
- escape-анализ;
- запахи кода (code smell);
- разыменование нулевого указателя;
- анализ зависимостей;
- статические анализаторы общего назначения;
- вспомогательный статический анализ.

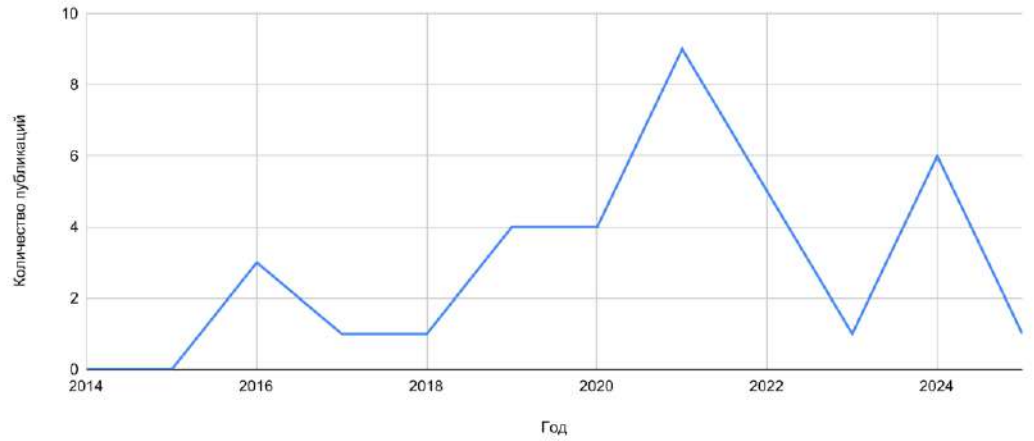


Рис. 1. Количество публикаций по годам.
Fig. 1. Number of papers split by year.

Табл. 2. Основные цели публикаций по статическому анализу языка Go.
Table 2. Goals of Golang static analysis papers.

Цель	Публикации
Параллелизм в Go	Lange [32], Ng [30], Lange [33], Veileborg [52], Scalas [35], Dilley [34], Khatchadourian [50], Gabet [38], Zhang [46], Wolf [48], Dilley2021 [45], Liu [49], Goguard [57]
Запахи кода (code smell)	Yasir [37], Sarker [47], Bergersen [31]
Неправильное использование криптографических библиотек	Li [51], Zhang [58]
Неправильное использование пакета unsafe	Lauinger [40], Costa [41]
Escape-анализ	Wang [39], Ding [61]
Разыменование нулевого указателя	Fu [59], Svace [56]
Статические анализаторы общего назначения	Svace [42-44, 53-56]
Анализ зависимостей	Cesarano [60]
Ошибки в CGO коде	Chen [62]
Вспомогательный статический анализ	Bodden [29], Wang [36]

5.1.1 Параллелизм в Go (Go concurrency)

Go – это статически типизированный язык, предназначенный для эффективного и надежного параллельного программирования. Язык предлагает описывать параллельные вычисления через встроенные в язык механизмы.

Горутины. Горутина [68] – это легковесный поток по сравнению с потоком (thread/тред/ветвь) операционной системы. За управление горутин отвечает библиотека времени выполнения Go, а не операционная система; библиотека мультиплексирует

множество горутин на меньшее количество потоков операционной системы. Важно отметить, что горутины, как и треды, взаимодействуют с общей памятью. Это означает, что все горутины имеют доступ к одним и тем же переменным и данным программы. Поэтому при работе с горутинами необходимо помнить о синхронизации доступа к общим данным.

Каналы. Для синхронизации с помощью передачи сообщений в Go существует встроенный примитив – канал [69-70], обеспечивающий безопасное взаимодействие между горутинами.

Механизмы синхронизации. В Go для синхронизации с использованием общей памяти используются традиционные примитивы, встречающиеся и в других языках программирования: мьютексы (sync.Mutex), которые используются для взаимного исключения доступа к общим ресурсам, предотвращая одновременное изменение данных несколькими горутинами; мьютексы чтения-записи (sync.RWMutex); механизм ожидания группы горутин sync.WaitGroup; атомарные операции (sync/atomic).

Данная особенность языка Go делает актуальным изучение уязвимостей, связанных с параллельностью, и методов поиска соответствующих ошибок. В нашем обзоре мы включили 12 публикаций, относящихся к этому направлению, среди них есть работы, направленные на изучение ошибок, чаще всего встречающихся в реальных проектах, а также на подходы к статическому анализу параллельных программ.

Проблемы, которые решаются в изученных работах, можно разделить на три вида (в табл. 3 показано разделение работ по этим видам):

- ошибки синхронизации с помощью передачи сообщений;
- ошибки синхронизации с использованием общей памяти;
- комбинированные работы.

Табл. 3. Цели публикаций по статическому анализу параллелизма в Go.
Table 3. Goals of papers devoted to static analysis of Go parallel features.

Цель	Публикации
Ошибки с каналами (передачей сообщений)	Lange [32], Ng [30], Lange [33], Veileborg [52], Scalas [35]
Ошибки с примитивами синхронизации	Khatchadourian [50]
Комбинированные работы	Gabet [38], Zhang [46], Wolf [48], Liu [49], Goguard [57]
Обзор	Dilley [34]

В Go рекомендуется передавать сообщения по каналам как средствам взаимодействия, наименее подверженным ошибкам. Эмпирическое исследование на открытых проектах, проведенное в 2019 году [71], показывает, что передача сообщений также подвержена ошибкам, как и общая память, и что неправильное использование каналов с еще большей вероятностью приведет к ошибкам взаимной блокировки (*deadlocks*, когда две или более горутины блокируют друг друга, ожидая ресурс, занятый другой), чем неправильное использование мьютексов. Большинство ошибок блокировки, вызванных синхронизацией с общей памятью, имеют те же причины и пути исправления, что и в традиционных языках программирования (Java, C++/C).

Но, например, реализация `sync.RWMutex` отличается от `pthread_rwlock_t` в языке C. В Go операции записи (*write lock*) имеют более высокий приоритет, чем операции чтения (*read lock*). Это означает, что если есть запрос на запись, то новые запросы на чтение будут отложены до завершения операции записи. В языке C в `pthread_rwlock_t` приоритет по умолчанию отдается операциям чтения. Это может привести к различиям в поведении, включая возможность взаимной блокировки в Go, которая невозможна в аналогичных сценариях в C. В исследовании пришли к выводу, что ошибки передачи сообщений чаще

всего связаны с неправильным закрытием каналов или неверной обработкой закрытых каналов; ошибки доступа к разделяемой памяти чаще всего возникают из-за неправильного использования мьютексов.

В 2019 было проведено другое исследование [34] использования механизма передачи сообщений в 865 открытых проектах на Go (с GitHub [67]). Исследование показало, что большинство проектов использует каналы, причем обычно применяются относительно простые паттерны взаимодействия (отправка в канал и получение сообщение из него).

Одна из первых работ [30] по поиску ошибок взаимной блокировки написана в 2016 году, где авторы предлагали искать такие ошибки с помощью проверки глобального графа сессий (*Global session graph* – объединяет все взаимодействия в программе в единый граф, который описывает общий протокол коммуникации). В публикации под типами сессий [72] понимаются операции Go, используемые для коммуникации между горутинами (например, чтение из канала или запись в него). Далее этой же группой исследователей в работах [32-33, 38] было предложено свое промежуточное представление для анализа многопоточных программ в виде поведенческих типов (*behavioral types*) и проверка такой модели [73-74] с помощью mCRL2 [75].

Другая часть работ стремится к полноте анализа, чтобы пропустить как можно меньше связанных с параллельностью ошибок. В статье [45] предложен метод ограниченной проверки моделей программы, который использует поведенческие типы и трансляцию в Promela [76] для верификации через Spin [77]. В рамках данной работы был разработан инструмент GOMELA [78]. Предложенный подход отличается поддержкой параметризованных программ, комбинированным анализом каналов, мьютексов и sync.WaitGroup, а также направлен на выявление ошибок взаимной блокировки, гонок данных и некорректного использования каналов.

В работе [50] описан простой внутрипроцедурный анализ потока данных для поиска ошибок, связанных с некорректным использованием мьютексов. Простой анализ работает быстрее и легче масштабируется. Есть ложноположительные срабатывания, связанные с отсутствием межпроцедурности детектора и проблемами в алгоритме.

В работе [49] описывается инструмент GCatch [79]. Он объединяет несколько статических детекторов ошибок, в том числе новый детектор блокирующих ошибок (BMOC). Для каждого канала `ch` в программе GCatch находит множество операций, в которых используется данный канал. Далее GCatch определяет набор горутин, обращающихся к каналу `ch`, путём анализа всех горутин, созданных в пределах области анализа канала `ch`. GCatch вычисляет комбинации путей, перебирая все возможные пути выполнения для каждой горутины. Детектор чувствителен к путям и использует SMT-решатель для определения того, приводят ли они к блокирующим ошибкам.

В статье [46] описывается инструмент GoDetector для обнаружения ошибок параллелизма в Go, основанный на анализе АСД. Инструмент анализирует только операции, связанные с каналами, а также игнорирует вызовы функций. Инструмент минимизирует ложноположительные срабатывания за счет анализа мертвого кода. Для каждой переменной `sync.WaitGroup` GoDetector использует конечный автомат с магазинной памятью, чтобы обнаружить ошибку, связанную с неправильным использованием примитива.

В работе [52] для поиска ошибок работы с каналами используется локализованная абстрактная интерпретация (*Localized Abstract Interpretation*), при которой программа разбивается на фрагменты, где фрагмент – это все операции с конкретным каналом. На этапе абстрактной интерпретации анализируется каждый фрагмент программы и строится граф суперлокаций (*superlocation graph*) – конечная система переходов, моделирующая пространство состояний фрагмента. На этапе обнаружения блокирующих ошибок выполняется обход графов суперлокаций с целью поиска абстрактных потоков (*abstract threads*) (множество горутин, встречающиеся в программе) в точках операций с каналами,

для которых отсутствует возможный путь разблокировки. Такие потоки во время выполнения программы могут блокироваться бесконечно. Данный подход не является полным или консервативным, но позволяет анализировать большие программы.

В статье [57] описывается инструмент статического анализа GoGuard, предназначенный для обнаружения блокирующих ошибок. Программа преобразуется в абстрактное представление (*Resource Flow Graph*, RFG), где каждая операция с каналами или мьютексами анализируется на предмет потенциальных блокировок. Ресурс в данном графе представляет каждый канал или объект мьютекса. В этом представлении используется концепция производитель-потребитель (*producer-consumer*) для каждого ресурса, чтобы моделировать поток его коммуникаций между горутинами, например, отправку и получение сообщений в канале или из канала. Ключевая идея заключается в том, что потребитель всегда блокирует выполнение горутины, если только производитель того же ресурса не сможет с ним сопоставиться. Далее RFG обходится, начиная с функции main, пока не встретится ситуация, в которой отсутствует соответствующий потребитель для производителя или наоборот. В таких случаях приостанавливается обход узла внутри его горутины и продолжается обход в других горутинах. После завершения процедуры все приостановленные горутины считаются заблокированными, что указывает на некорректное использование примитивов параллелизма.

Резюме. Go предоставляет несколько механизмов для работы с параллелизмом: горутины, каналы, мьютексы, `sync.WaitGroup`, `sync.Once`. Среди основных ограничений анализа, присущих почти каждой работе, можно выделить: проблемы масштабируемости, наличие ложноположительных срабатываний, покрытие не всех примитивов, а только ограниченного их количества (например, игнорирование `sync.WaitGroup`, `sync.Once` приводит к ложным срабатываниям). Среди параллельных ошибок чаще всего встречаются простые ошибки, которые не требуют сложного статического анализа. Чтобы находить более сложные ошибки, необходимо учитывать все примитивы параллельного программирования в коде и моделировать их взаимодействие, использовать чувствительный к путям и полям структур анализ. Чтобы увеличить скорость анализа, анализируется не весь граф потока управления (ГПУ), а только его подмножество, связанное с примитивами (каналами, мьютексами). Основная проблема статического анализа с помощью построения модели программы, использующей параллельные примитивы, и ее последующей верификации – это масштабируемость на большие проекты.

5.1.2 Неправильное использование криптографических библиотек

В данном разделе обсуждается некорректное использование криптографических библиотек (в том числе устаревших версий), что может угрожать безопасности проекта. Такие ошибки – часть более широкого класса ошибок, неправильного использования программных интерфейсов (Application Programming Interface, API), для которого в англоязычной литературе используется термин API misuse. Существующие инструменты поиска этих библиотек основаны на черных или белых списках.

Первая статья, написанная в 2022 году [51], описывает работу автоматического инструмента CryptoGo, предназначенного для обнаружения проблем безопасности в криптографических библиотеках (`crypto/...` [80] и `golang.org/x/crypto/...` [81]), используемых проектами на языке Go. Инструмент использует прямой и обратный *анализ помеченных данных* (*Taint Analysis*) и 12 криптографических правил. В анализе помеченных данных используются четыре типа функций для выявления опасных потоков: источники (*sources*), распространители (*propagators*), фильтры (также называемые санитайзерами, *sanitizers*) и приемники (*sinks*). В частности, источник – это функция, которая генерирует ненадёжный ввод; приемник – функция, принимающая ненадёжный ввод и передающая его в чувствительное место назначения. Распространитель – это функция, которая передаёт

ненадёжные данные из одной части программы (через переменную) в другую. Фильтр – функция, проходя через которую, переменная становится безопасной. Результат работы был протестирован на 120 открытых проектах, использующих криптографические библиотеки: CryptoGo выявил 622 предупреждения неправильного использования API (с точностью 95,5%) и обнаружил, что 83,33% проектов Go имеют хотя бы одно такое использование.

Ограничение работы [51]. Реализовано покрытие только официального API. CryptoGo может выдать ложноотрицательные результаты в случае вызова API из неофициальных криптографических библиотек Go, а также ложноположительные результаты из-за отсутствия чувствительности к путям.

В работе [58] описывается автоматический инструмент обнаружения неправильного использования криптографических библиотек (`crypto/`... [80] и `golang.org/x/crypto/`... [81]) – Gopher. Gopher состоит из двух основных компонентов: CryDict и Detector. CryDict переводит криптографические правила в формальные ограничения; Detector выявляет неправильные использования криптографических библиотек на основе этих ограничений и предоставляет необходимую информацию о потоке данных CryDict для дальнейшего вывода и разработки новых ограничений.

Ограничение работы [58]. Ложноположительные срабатывания из-за ошибок в описании ограничений, отсутствия чувствительности к путям, построения графа вызовов с помощью анализа иерархии классов. Ложноотрицательные срабатывания из-за возможности анализировать файлы только на языке Go; кроме того, не поддерживаются формальные ограничения для неофициальных (пользовательских) криптографических библиотек.

Резюме. Поиск ошибок, связанных с криптографическими библиотеками, выполняется с помощью анализа помеченных данных, чувствительности к путям и генерации формальных ограничений (например, черные или белые списки; спецификации; пометка источников и приемников) для этих библиотек. Основная трудность – наличие неофициальных библиотек, для которых необходимой информации нет.

5.1.3 Неправильное использование пакета unsafe

Язык программирования Go обеспечивает безопасность памяти и потоков через автоматическое управление памятью (сборки мусора) и строгую систему типов, однако использование пакета `unsafe` позволяет обойти эти механизмы безопасности, что может привести к уязвимостям, таким как переполнение буфера или повреждение памяти.

В статье [40] было исследовано 500 открытых проектов на GitHub [67] и найдено 1400 использований пакета `unsafe`. Среди этих проектов 38% проектов используют `unsafe` напрямую, 91% имеют зависимости, которые используют `unsafe`.

В работе были определены основные паттерны использования пакета `unsafe`:

- приведение типов (например, строки к байтам);
- управление памятью (например, использование `uintptr` для арифметики указателей);
- оптимизация производительности (например, уменьшение числа лишних копирований данных).

Здесь также определены основные потенциальные уязвимости:

- гонки сборщика мусора: неправильное использование `uintptr` может привести к освобождению памяти, которая все еще используется;
- ошибки escape-анализа: неправильное использование указателей может привести к утечкам памяти или повреждению данных;
- переполнение буфера: неправильное приведение типов может привести к доступу за пределами выделенной памяти.

В работе также был предложен АСД-детектор [82] go-safer, который ищет некорректные использования `reflect.SliceHeader`, `reflect.StringHeader` и небезопасные преобразования между структурами с архитектурно-зависимыми типами полей. Важно отметить, что в 2025 году структуры `reflect.SliceHeader` и `reflect.StringHeader` являются устаревшими (deprecated), так как, начиная с версии Go 1.20, вместо них появились функции `unsafe.String` и `unsafe.StringData`; `unsafe.Slice` и `unsafe.SliceData` соответственно.

Исследование [41] изучает использование пакета `unsafe` в реальных проектах. В статье авторы проанализировали 2438 популярных открытых проектов на Go. Исследование показывает, что хотя `unsafe` широко используется для производительности и интеграции, этот пакет также создает значительные риски и проблемы, требующие улучшенных инструментов и практик для их устранения.

Резюме. Ошибки, связанные с использованием пакета `unsafe`, могут приводить к серьёзным проблемам, включая нарушения целостности памяти и неопределённое поведение. В одной из рассмотренных работ небезопасные преобразования обнаруживались с помощью анализа АСД. Однако в ходе обзора мы не выявили публикаций, посвящённых систематическому анализу современного API `unsafe` в языке Go.

5.1.4 Запахи кода

Запахи кода (*code smells*)–синтаксически корректные участки кода, которые дают основание заподозрить проблемы с архитектурой приложения или качеством кода, при этом, строго говоря, не являющиеся ошибками.

В магистерской диссертации [31] описывается статический анализ для поиска запахов кода. Инструмент поставляется в виде плагина для SonarQube [83]. В работе с помощью анализа АСД вычисляется цикломатическая сложность [84] функции и метода; небезопасное использование переменной цикла в горутине (потеряло актуальность после Go 1.22); игнорирование проверки error из функции и другие.

В работе [37] описывается реализация инструмента для поиска *God Struct* [85] (структура, выполняющая слишком много функций) в программах на языке Go. С помощью анализа АСД вычисляются три метрики: *WMC (Weighted Method Count), TCC (Tight Class Cohesion)* и *ATFD (Access To Foreign Data)*. Авторы успешно протестировали инструмент на двух открытых проектах.

В статье [47] описывается анализ архитектуры проектов на языке Go с использованием метрик *MOOD (Metrics for Object-Oriented Design)* [86] и *CK (Chidamber and Kemerer Metrics)* [87] – это два набора метрик, используемых для оценки качества объектно-ориентированного программного обеспечения. Они помогают анализировать архитектурные характеристики программного кода, такие как сложность, связность, наследование и инкапсуляция. Для обнаружения запахов кода в Go в работе эти метрики помогают выявить *God Struct* или *Feature Envy* (метод, использующий данные других структур). Описывается успешное тестирование на 5 открытых проектах. В статье большее внимание уделяется вычислению метрик, чем организации анализа.

Резюме. Ошибки типа запахи кода можно выявлять по различным метрикам: *MOOD, CK, WMC, TCC, ATFD*, которые могут использоваться как для популярных объектно-ориентированных языков программирования, так и для Go. Для их вычисления достаточно анализа АСД.

5.1.5 Разыменование нулевого указателя

В работе [59] с помощью анализа потока значений ищется ошибка разыменования нулевого указателя. На основе (*points-to*) [88] анализа в работе строится граф значений.

Авторы расширили этот алгоритм моделированием семантики Go-каналов.

Ограничения работы [59]. Алгоритм использует только анализ потока значений, что приводит к наличию ложноположительных срабатываний в параллельных программах. В качестве решения данной проблемы авторы рекомендуют использовать SMT-решатель, отмечая, что предложенный алгоритм будет работать медленнее. Отсутствие результатов тестирования на средних и больших проектах.

Статья [56] посвящена статическому анализу ассоциативных массивов (map) в языке Go для поиска разыменования нулевых указателей при извлечении ключей. Анализ использует символьное выполнение, на основе которого была промоделирован семантика Go map.

Ограничения работы [56]. Ложные срабатывания возникают из-за ограниченной точности при моделировании условий достижимости, когда анализатор не может полностью выявить взаимосвязи между переменными. Они также могут быть обусловлены сложной логикой программы, например, при итерации по всем ключам отображения без предварительной проверки их существования.

Резюме. В описанных работах используются методы анализа, применимые и для анализа других языков, но с расширением для специфичных конструкций языка Go: каналы (channels), ассоциативные массивы (maps).

5.1.6 Статический анализатор общего назначения

Ранее перечисленные публикации были сфокусированы только на какой-то одной задаче: поиск ошибок в параллельных программах, неправильное использование API, неправильное использование пакета unsafe и т.д. Для Go существуют как открытые инструменты (Go vet [2], Gosec [9]), так и закрытые (Svace, Coverity [89]), которые ищут сразу множество ошибок или самостоятельно, или агрегируя несколько инструментов (golangci-lint [12]).

Go vet, Gosec, golangci-lint – открытые анализаторы, ищут шаблоны на АСД, которые потенциально могут вызывать ошибки. Публикаций по особенностям анализа для данных инструментов мы не нашли.

В работах [42-43, 54], посвященных статическому анализу языка Go в инструменте Svace, описывается межпроцедурный статический анализ на основе резюме. В нем реализован движок символьного выполнения, с помощью которого инструмент может находить множество сложных ошибок: разыменование нулевого указателя, деление на ноль, целочисленные переполнения, утечки ресурсов и чувствительных данных и другие. Для проверки выполнимости путей в Svace используется SMT-решатель. Инструмент также отличается наличием анализов, посвященным особенностям языка Go, таким как анализ замыканий [53] и анализ ассоциативных массивов (maps) [56]. Стоит отметить, что для получения промежуточного представления Svace использует технику автоматической контролируемой сборки [55] (*build capture*), которая адаптирована для Go (отслеживаются как вызовы команды go build/install/run, так и прямые вызовы компилятора Go compile).

Coverity – закрытый инструмент статического анализа. По документации можно сделать вывод, что в инструменте есть движок межпроцедурного чувствительного к путям и контексту анализа, а также поддержка контролируемой сборки; используется ли это для Go – неизвестно.

Резюме. Для Go существуют как открытые легковесные анализаторы, выявляющие типичные ошибки на уровне АСД, так и закрытые тяжеловесные инструменты, использующие более сложные методы для обнаружения нетривиальных ошибок, такие как Svace и Coverity.

5.1.7 Escape-анализ

Работа [39] посвящена исследованию и улучшению escape-анализа (он помогает определить, должна ли переменная храниться на стеке (*stack*) или на куче (*heap*), если переменная может

использоваться после завершения функции – она размещается на куче) в языке Go. Авторы изучили работу этого алгоритма в компиляторе Go и предложили подходы для его улучшения: оптимизация для ситуаций, в которых переменная-указатель используется в качестве аргумента функции. Чтобы получить информацию об утекающих переменных, инструмент изучает журналы сборщика мусора (*garbage collection (GC)*), на основании чего оптимизирует исходный код программы. Затем с помощью анализа АСД- и LLVM-промежуточных представлений проверяется целостность памяти. Метод на основе АСД проверяет, является ли вызов функции синхронным. Указатели, передаваемые в синхронный вызов функции в качестве аргументов, не будут освобождены до завершения вызова, поскольку такой вызов не создаёт новую горутину (или поток). Следовательно, в этом случае оптимизация является корректной. Если вызов функции является асинхронным, то используется метод на основе LLVM-представления.

В публикации [61], посвященной улучшению escape-анализа в Go, авторы предлагают новый алгоритм, который использует чувствительность к полям и *points-to* анализ, чтобы повысить точность без значительного увеличения времени компиляции. Алгоритм реализован в фреймворке MEA, который использует промежуточное представление LLVM. Для улучшения межпроцедурного анализа в работе используется подход, основанный на резюме функций. Анализируются замыкания: в месте создания структуры замыкания и передачи в нее захваченных переменных вставляется псевдовызов замыкания, где используется его резюме (анализ делает консервативное предположение, что адрес захваченной переменной получается из кучи, тем самым выделяя захваченную переменную в куче). Также в работе анализируются инструкции defer и определяются функции, вызываемые через них. Затем для выявленных отложенных вызовов вставляются псевдовызовы в конце родительской функции в соответствии с их порядком в графе потока управления.

Резюме. Основная проблема escape-анализа в Go – его консервативность; часто переменные размещаются в куче, хотя могли бы остаться на стеке. К улучшению escape-анализа можно подойти с разных сторон: оптимизируя исходный код или создание нового алгоритма. В первой работе оптимизируется ситуация, когда переменная-указатель является формальным параметром функции, а во второй работе предлагается чувствительный к полям анализ.

5.1.8 Вспомогательный статический анализ

Статья [36] посвящена разработке инструмента Go-Sanitizer – генератора проверок (*assertions*), которые помогают выявлять и локализовывать ошибки с помощью дальнейшего фаззинга и модульного тестирования. Рекомендация проверок и вставка их в код проводится с помощью анализа АСД. Проверки генерируются для 9 типов ошибок (из CWE [90]: 128, 190, 191, 785, 466, 824, 478, 1077, 777). Например, для CWE-824 авторы используют проверку указателя на `nil`. Для проекта [91] было вставлено 42 проверки и обнаружено 12 ошибок, которые не удалось найти с помощью фаззинга или модульного тестирования.

В одной из первых работ [29] по статическому анализу Go описывается разработка анализа потоков данных, включающего анализ объектных типов (структуры, массивы) и коммуникации через каналы; также обсуждается, как эта информация может использоваться во время выполнения (инструментирование кода). Для точного отслеживания потоков данных используется комбинация методов статического анализа: контекстно-чувствительный межпроцедурный анализ, анализ алиасов, анализ помеченных данных.

Резюме. В данной секции рассматриваются работы, в которых статический анализ применяется как инструмент для извлечения дополнительной информации, используемой в дальнейшем при проведении динамического анализа, включая фаззинг, инструментирование кода и модульное тестирование.

5.1.9 Анализ зависимостей

Рост сложности программ привел к использованию сторонних библиотек (*third-party libraries*) для снижения затрат на разработку. Вместе с тем такие зависимости могут содержать дополнительные уязвимости [92]. Go изначально был разработан с поддержкой децентрализованных реестров, где зависимости могут быть импортированы напрямую из git-репозиториев [93]. Отсутствие централизованного реестра в Go ограничивает возможности по отслеживанию уязвимостей в модулях [94].

В статье [60] описывается инструмент GoSurf, предназначенный для облегчения анализа зависимостей с открытым исходным кодом и аудита с целью обнаружения вредоносного кода. В работе рассматриваются специфические для Go векторы атаки (12 векторов), например, статическая генерация кода (`//go:generate`), тестовые функции (запускаемые через `go test`), методы-конструкторы, интерфейсы, `unsafe` и другие. Эти векторы атаки GoSurf обнаруживает через анализ АСД программы. Анализатор был протестирован на 500 часто импортируемых Go модулях, было выявлено проявление некоторых из 12 векторов атак в каждом из проанализированных модулей.

Резюме. Перед тем, как интегрировать стороннюю зависимость, разработчики должны провести тщательный аудит безопасности, сделать это можно с помощью статического анализа. Для Go можно выделить специфичные векторы атак, чтобы их обнаружить, достаточно анализа АСД.

5.1.10 Ошибки в CGO-коде

CGO – интерфейс внешних функций в Go, обеспечивающий взаимные вызовы функций между Go и Си, позволяющий компонентам среды выполнения Go, таким как сборщик мусора, аллокатор памяти и планировщик горутин, взаимодействовать с программами на Си во время выполнения.

В исследовании [62] авторы представили первое исследование по использованию CGO-кода в открытых проектах на GitHub [67]. Они разработали статический анализатор CGOAnalyzer, работающий над АСД представлении, и применили анализатор для определения основных шаблонов использования CGO.

Резюме. Инструментов статического анализа кода, ищущих ошибки неправильного использования CGO, мы не обнаружили. Данное направление открыто для разработок.

5.1.11 Типы ошибок

На основе анализа публикаций, попавших в исследование, мы выделили 10 направлений использования статического анализа, а также следующие основные типы ошибок (без учета анализаторов общего назначения):

- неправильное использование каналов, горутин и примитивов синхронизации;
- запахи кода;
- неправильное использование криптографических библиотек;
- неправильное использование пакета unsafe;
- разыменование nil;
- векторы атак;
- неправильное использование CGO.

Вопрос 1: Мы выделили 9 направлений публикаций по статическому анализу Go и 7 типов ошибок. Статический анализ чаще всего проводится, чтобы находить ошибки, связанные с параллельным программированием.

5.2 Реализация анализа

5.2.1 Промежуточное представление

В табл. 4 перечислены промежуточные представления, которые чаще всего используются инструментами статического анализа кода. Чаще всего используется АСД, предоставляемое стандартной библиотекой Go – `go/ast`, для поиска определенных шаблонов дерева, например, блоков `unsafe` [40] кода или CGO [62].

Вторым по популярности является SSA-представление [95]. Создатели языка Go специально для статического анализа разработали свое SSA-представление (`golang.org/x/tools/go/ssa`), генерируемое инструментом `ssadump` [96]. Некоторые работы на основе стандартного SSA-представления языка Go строят свое собственное промежуточное представление (например, SvaceIR [43] – низкоуровневое типизированное промежуточное представление).

Несколько работ выбрали LLVM-представление [97] компилятора Go, развиваемого компанией Google в рамках проекта gollvm [98]. В работе [61] отмечается, что алгоритмы, разработанные на основе LLVM-представления, легче адаптировать для других языков программирования. Это связано с тем, что LLVM предоставляет универсальное промежуточное представление и уже включает набор инструментов для анализа.

В другой части работ для реализации анализа авторы использовали собственный парсер языка Go, строящий SSA-представление. Выбор в пользу SSA-представления устраняет сложности, связанные с изменением значений переменных в процессе выполнения программы, так как в SSA каждой переменной значение присваивается только один раз, что значительно упрощает анализ потока данных. SSA позволяет выполнять такие анализы, как символьное выполнение, абстрактная интерпретация и анализ потока данных, межпроцедурный анализ, чувствительный к путям и контексту анализ, которые трудно реализовать на уровне АСД. SSA облегчает обработку сложных языковых конструкций, например, в АСД замыкания могут быть представлены как вложенные функции, но их анализ требует дополнительной работы для отслеживания захваченных переменных, в SSA же захваченные переменные явно представлены через их адреса, что упрощает анализ. АСД, в свою очередь, больше подходит для синтаксического анализа.

Табл. 4. Промежуточное представление, используемое для статического анализа.
Table 4. Intermediate representations utilized for static analysis.

Промежуточное представление	Публикации
go/ast	Wang [36, 39], Bergersen [31], Lauinger [40], Zhang [46], Chen [62], Khatchadourian [50], Sarker [47], Dilley2019 [34], Dilley2021 [45], Yasir [37], GoSurf [60], Costa [41]
golang.org/x/tools/go/ssa	Bodden [29], Lange [32, 33], Liu [49], Li [51], Veileborg [52], Zhang [58], Svace [42–44, 53–56]
Gollvm IR	Wang [39], Fu [59], Ding [61]
Собственный парсер	Ng [30], Gabet [38]
Явно не указано	Scalas [35], Gobra[48], Liu2024 [57]

Вопрос 2.1: Чаще всего для статического анализа программ на языке Go используются промежуточные представления АСД- или SSA-, предоставляемые экосистемой Go: `go/ast` и `golang.org/x/tools/go/ssa`.

5.2.2 Методы анализа

Во всех рассмотренных публикациях анализируется поток управления или данных программы, а также используются специальные методы статического анализа. Для анализа параллельных программ чаще всего используется метод верификации типов и модели (*Type/Model Checking*), с помощью которого анализируется поведение программы и проверяется ее корректность [30, 32-33, 38, 45, 71]. Также в некоторых работах используется символьное выполнение (например, в работе [58] для вывода формальных ограничений, в анализаторе Svace внутрипроцедурный анализ основан на символьном выполнении с объединением состояний в точках слияния путей). В работе [52] используется локализованная абстрактная интерпретация для поиска ошибок при работе с каналами в Go. Для отслеживания потока помеченных данных применяется анализ помеченных данных (в работах [29, 44, 51]). Во многих работах комбинируются сразу несколько методов статического анализа (например, в работах [43, 49, 58, 61]).

Вопрос 2: Основные методы статического анализа, используемые для анализа программ на Go, включают верификацию типов и моделей, символьное выполнение, анализ потока данных, анализ помеченных данных. Часто для повышения точности анализа методы комбинируются.

Отметим, что для языка Go общие методы статического анализа работают или без изменений, или со сравнительно небольшими вариациями для учета специфики языка (каналы, замыкания, инструкции defer, коллекции). Это позволяет реализовывать поддержку Go в многоязыковых анализаторах, в частности, в анализаторе Svace.

Для отсеивания несуществующих путей в статическом анализаторе может использоваться SMT-решатель [49, 99]. Для того, чтобы разрешить вызов процедур для указателей на функции в Go, можно использовать девиртуализацию [54] или построение среза программы (извлекается подмножество кода программы (срез), которое влияет на определенные переменные или операции). Например, в работе [58] срезы используются для анализа иерархии структур в Go. Также для Go применимы техники анализа потока данных (например, анализ недостижимого кода [43, 46]). Часть работ для межпроцедурного анализа используют резюме (например, [43, 61]). Для моделирования поведения библиотечных функций в Go в работах [43, 56] используются спецификации (модели функций, представляющие из себя код на анализируемом языке, в котором используются функции для обозначения некоторых свойств, например, фиксируется, что функция создает или освобождает ресурсы).

Вопрос 2.2: Чаще всего при статическом анализе программ на языке Go принимаются во внимание примитивы параллельного программирования, встраивание структур, замыкания, инструкция `defer` в Go.

Большая часть работ посвящена одной особенности языка Go: примитивам параллельного программирования, встраиванию структур (*Struct Embedding* [54], коллекциям в Go ([56]), замыканиям и инструкции отложенного вызова defer [53].

5.2.3 Ограничения работ

Задача выявления всех ошибок с помощью статического анализа в программе является алгоритмически неразрешимой [100]. Это означает, что невозможно гарантированно найти все ошибки определённого типа в произвольной программе без ложных срабатываний.

К ложноположительным срабатываниям (*false positive*, то есть ошибкам первого рода, когда выдалось ложное предупреждение об ошибке) приводят следующие причины: отсутствие чувствительности к путям [51-52, 58]; отсутствие чувствительности к потоку (например, [53]); сложность ГПУ [52]; реализация метода (например, учитываются только неизменяемые переменные, не учитываются конструкции внутри структур данных, бесконечные состояния

[33]), виртуальные методы [45], ограничения пакетов анализа (анализ алиасов, анализ графа вызовов) [49], отсутствие межпроцедурности [49], не все конструкции языка учитываются (например, могут не учитываться конструкции `Once` и `defer` [46]).

К ложноотрицательным срабатываниям (*false negative*, то есть ошибкам второго рода, когда ошибка в программе есть, но она не была обнаружена анализатором) приводят следующие причины: реализация метода – отсутствие необходимых данных о пользовательских библиотеках [51, 58]; недостаточная масштабируемость анализа (например, [30, 32-33, 46, 50]), неполный учет ошибочных паттернов [49, 52], неполный анализ межпроцедурных связей [52], неполный учет конструкций языка [32, 46] (например, не учитываются конструкции `Mutex`, `WaitGroup` и `Cond` [52]).

Вопрос 2.3: Чаще всего инструменты статического анализа, описанные в публикациях, сталкиваются с проблемами масштабируемости, ограничений в реализации методов анализа, отсутствия чувствительности к путям или потоку.

6. Ограничения данного обзора

В обзор включены публикации за период с 2012 по 2025 год. Для части инструментов с закрытым исходным кодом не удалось найти публикации с подробным описанием реализуемого анализа; однако известно, что эти инструменты поддерживают язык Go.

В исследовании учитываются публикации только на английском и русском языках. Выбор ключевых слов, интерпретация результатов и классификация публикаций могут зависеть от предпочтений авторов, что снижает объективность обзора.

7. Заключение

Чтобы перечислить задачи, стоящие перед разработчиками статических анализаторов, мы выполнили обзор публикаций о подходах, связанных с использованием статического анализа в программах на языке Go. В процессе этого обзора мы изучили 34 научных статьи, опубликованные на конференциях и в журналах по разработке программного обеспечения, языкам программирования и безопасности. Мы ставили себе целью изучить задачи, на которые нацелен статический анализ; основные методы статического анализа, используемые в публикациях; реализации самого анализа и его ограничений.

Наш обзор помог сделать следующие ключевые выводы:

- на основе найденных публикаций мы выделили 9 направлений использования статического анализа для Go;
- больше всего публикаций посвящено поиску ошибок взаимной блокировки и ошибкам с использованием традиционных примитивов синхронизации (мьютексов);
- чаще всего для статического анализа программ на языке Go используются АСД- или SSA- промежуточные представления;
- основные методы статического анализа, используемые для анализа программ на Go, включают верификацию типов и моделей, символьное выполнение, анализ потока данных, анализ помеченных данных;
- примитивы параллельного программирования, встраивание структур, замыкания, defer в Go чаще всего принимаются во внимание при статическом анализе программ;
- инструменты статического анализа, описанные в публикациях, сталкиваются с проблемами масштабируемости, ограничений в реализации методов анализа, отсутствия чувствительности к путям или потоку;
- в целом, методы статического анализа для языка Go такие же, как и для анализа других языков программирования, с корректировкой на особенности языка.

Надеемся, что данный обзор литературы поможет как разработчикам статических анализаторов, так и разработчикам программ на языке Go.

Список литературы / References

[1]. Golang main page. https://go.dev/. Доступ: 2024-01-10.

[2]. Go vet main page. https://golang.org/cmd/vet/. Доступ: 2023-10-01.

[3]. Go tools. https://godoc.org/golang.org/x/tools. Доступ: 2023-10-05.

[4]. A. A. Donovan и B. W. Kernighan. The Go programming language. Addison-Wesley Professional, 2015.

[5]. Effective go– the go programming language. https://go.dev/doc/effective_go. Доступ: 2024-10-05.

[6]. Y. Feng и Z. Wang. Towards understanding bugs in Go programming language. В 2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS), страницы 284–295, 2024. DOI: 10.1109/QRS62785.2024.00036.

[7]. J. Wu и J. Clause. Assessing Golang static analysis tools on real-world issues. Available at SSRN 5208109.

[8]. Errcheck main page. https://github.com/kisielk/errcheck. Доступ: 2024-01-10.

[9]. Go security checker – gosec. https://github.com/securego/gosec. Доступ: 2024-01-10.

[10]. Revive. https://github.com/mgechev/revive. Доступ: 2025-02-11.

[11]. Staticcheck main page. https://staticcheck.io. Доступ: 2024-01-10.

[12]. Go linters runner– golangci-lint. https://github.com/golangci/golangci-lint. Доступ: 2023-10-01.

[13]. Go developer survey 2024 h2 results. https://go.dev/blog/survey2024-h2-results. Доступ: 2024-12-21.

[14]. Gopls. https://pkg.go.dev/golang.org/x/tools/gopls. Доступ: 2023-10-03.

[15]. Visual studio code. https://code.visualstudio.com/. Доступ: 2023-10-04.

[16]. E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek и J. Vitek. On the impact of programming languages on code quality: a reproduction study. ACM Transactions on Programming Languages and Systems (TOPLAS), 41(4):1–24, 2019.

[17]. M. H. Ruge. Analysis of software engineering automation tools for Go. Universidad de los Andes. https://hdl.handle.net/1992/54945. Доступ: 2025-02-11.

[18]. Opennota/check. https://gitlab.com/opennota/check. Доступ: 2025-02-11.

[19]. M. bohusl'avek, mibk/dupl. https://github.com/mibk/dupl. Доступ: 2025-02-11.

[20]. Fzipp, fzipp/gocyclo. https://github.com/fzipp/gocyclo. Доступ: 2025-02-11.

[21]. A. kohler, alexkohler/prealloc. https://github.com/alexkohler/prealloc. Доступ: 2025-02-11.

[22]. Stripe/safesql. https://github.com/stripe/safesql. Доступ: 2025-02-11.

[23]. Gofmt. https://pkg.go.dev/cmd/gofmt. Доступ: 2023-10-04.

[24]. Delve. https://github.com/go-delve/delve. Доступ: 2025-02-11.

[25]. Godoc. https://pkg.go.dev/golang.org/x/tools/cmd/godoc. Доступ: 2025-02-11.

[26]. B. Kitchenham. Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004):1–26, 2004.

[27]. L. Li, T. F. Bissyand'e, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein и L. Traon. Static analysis of android apps: a systematic literature review. Information and Software Technology, 88:67– 5, 2017. DOI: https://doi.org/10.1016/j.infsof.2017.04.001.

[28]. P. H. Nguyen, M. Kramer, J. Klein, M. Schulz, B. R. de Supinski и M. S. M..uller. An extensive systematic review on the model-driven development of secure systems. Scientific Programming, 21(3-4):109–121, 2013.

[29]. E. Bodden, K. I. Pun, M. Steffen, V. Stolz и A.-K. Wickert. Information flow analysis for go. В International Symposium on Leveraging Applications of Formal Methods, страницы 431–445. Springer, 2016.

[30]. N. Ng и N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. В Proceedings of the 25th International Conference on Compiler Construction, страницы 174–184, 2016.

[31]. C. B. Bergersen. Detection of Bugs and Code Smells through Static Analysis of Go Source Code. Дис. маг., 2016.

[32]. J. Lange, N. Ng, B. Toninho и N. Yoshida. Fencing off go: liveness and safety for channel-based programming. ACM SIGPLAN Notices, 52(1):748–761, 2017.

[33]. J. Lange, N. Ng, B. Toninho и N. Yoshida. A static verification framework for message passing in go using behavioural types. В Proceedings of the 40th International Conference on Software Engineering, страницы 1137–1148, 2018.

[34]. N. Dilley и J. Lange. An empirical study of messaging passing concurrency in go projects. В 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), страницы 377–387. IEEE, 2019.

[35]. A. Scalas, N. Yoshida и E. Benussi. Verifying message-passing programs with dependent behavioural types. В Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, страницы 502–516, 2019.

[36]. C. Wang, H. Sun, Y. Xu, Y. Jiang, H. Zhang и M. Gu. Go-sanitizer: bug-oriented assertion generation for Golang. В 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), страницы 36–41. IEEE, 2019.

[37]. R. M. Yasir, M. Asad, A. H. Galib, K. K. Ganguly и M. S. Siddik. Godexpo: an automated god structure detection tool for golang. В 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), страницы 47–50. IEEE, 2019.

[38]. J. Gabet и N. Yoshida. Static race detection and mutex safety and liveness for go programs. В 34th European Conference on Object-Oriented Programming (ECOOP 2020), страницы 4–1. Schloss Dagstuhl–Leibniz-Zentrum f..ur Informatik, 2020.

[39]. C. Wang, M. Zhang, Y. Jiang, H. Zhang, Z. Xing и M. Gu. Escape from escape analysis of Golang. В Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, страницы 142–151, 2020.

[40]. J. Lauinger, L. Baumg..artner, A.-K. Wickert и M. Mezini. Uncovering the hidden dangers: finding unsafe go code in the wild. В 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), страницы 410–417. IEEE, 2020.

[41]. D. E. Costa, S. Mujahid, R. Abdalkareem и E. Shihab. Breaking type safety in Go: an empirical study on the usage of the unsafe package. IEEE Transactions on Software Engineering, 48(7):2277–2294, 2021.

[42]. I. Bolotnikov и A. Borodin. Interprocedural static analysis for finding bugs in go programs. Programming and Computer Software, 47:344–352, 2021.

[43]. A. Borodin, V. Dvortsova, S. Vartanov и A. Volkov. Static analyzer for Go. В 2021 Ivannikov Ispras Open Conference (ISPRAS), страницы 17–25. IEEE, 2021.

[44]. А. Е. Бородин, А. В. Горемыкин, С. П. Вартанов и А. А. Белеванцев. Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе svace. Труды Института системного программирования РАН, 33(1):7–32, 2021.

[45]. N. Dilley и J. Lange. Automated verification of go programs via bounded model checking (artifact), 2021.

[46]. D. Zhang, P. Qi и Y. Zhang. Godetector: detecting concurrent bug in go. IEEE Access, 9:136302–136312, 2021.

[47]. M. K. Sarker, A. A. Jubaer, M. S. Shohrawardi, T. C. Das и M. S. Siddik. Analysing GoLang projects' architecture using code metrics and code smell. В Proceedings of the First International Workshop on Intelligent Software Automation: ISEA 2020, страницы 53–63. Springer, 2021.

[48]. F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira и P. M..uller. Gobra: modular specification and verification of go programs. В International Conference on Computer Aided Verification, страницы 367–379. Springer, 2021.

[49]. Z. Liu, S. Zhu, B. Qin, H. Chen и L. Song. Automatically detecting and fixing concurrency bugs in go software systems. В Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, страницы 616–629, 2021.

[50]. F. T. H. M. R. Khatchadourian и Y. Cong. How many mutex bugs can a simple analysis find in Go programs? В Annual Conference of the Japanese Society for Software Science and Technology, 2022.

[51]. W. Li, S. Jia, L. Liu, F. Zheng, Y. Ma и J. Lin. Cryptogo: automatic detection of go cryptographic api misuses. В Proceedings of the 38th Annual Computer Security Applications Conference, страницы 318–31, 2022.

[52]. O. H. Veileborg, G.-V. Saioc и A. Møller. Detecting blocking errors in go programs using localized abstract interpretation. В Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, страницы 1–12, 2022.

[53]. A. Borodin, V. Dvortsova и A. Volkov. Interprocedural static analysis for Go with closure support. В 2022 Ivannikov Ispras Open Conference (ISPRAS), страницы 1–6. IEEE, 2022.

[54]. A. Galustov, A. Borodin и A. Belevantsev. Devirtualization for static analysis with low level intermediate representation. В 2022 Ivannikov Ispras Open Conference (ISPRAS), страницы 18–23. IEEE, 2022.

[55]. V. Dvortsova, A. Izbyshev, A. Borodin и A. Belevantsev. Static analysis for Go: build interception. В 2023 Ivannikov Ispras Open Conference (ISPRAS), страницы 52–57. IEEE, 2023.

[56]. Д. Н. Субботин, А. Е. Бородин и В. В. Дворцова. Статический анализ ассоциативных массивов в Go. Труды Института системного программирования РАН, 36(3):21–34, 2024.

[57]. B. Liu и D. Joshi. Goguard: efficient static blocking bug detection for Go. В International Static Analysis Symposium, страницы 216–241. Springer, 2024.

[58]. Y. Zhang, B. Li, J. Lin, L. Li, J. Bai, S. Jia и Q.Wu. Gopher: high-precision and deep-dive detection of cryptographic api misuse in the go ecosystem. В Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, страницы 2978–2992, 2024.

[59]. S. Fu и Y. Liao. Golang defect detection based on value flow analysis. В 2024 9th International Conference on Electronic Technology and Information Science (ICETIS), страницы 358–363. IEEE, 2024.

[60]. C. Cesarano, V. Andersson, R. Natella и M. Monperrus. Gosurf: identifying software supply chain attack vectors in Go. arXiv preprint arXiv:2407.04442, 2024.

[61]. B. Ding, Q. Li, Y. Zhang, F. Tang и J. Chen. Mea2: a lightweight field-sensitive escape analysis with points-to calculation for Golang. Proceedings of the ACM on Programming Languages, 8(OOPSLA2):1362–1389, 2024.

[62]. J. Chen, B. Ding, Y. Zhang, Q. Li и F. Tang. An empirical study of Cgo usage in Go projects–distribution, purposes, patterns and critical issues. Purposes, Patterns and Critical Issues.

[63]. Google scholar. https://scholar.google.com. Доступ: 2025-02-11.

[64]. Ieee xplore. https://ieeexplore.ieee.org/Xplore/home.jsp. Доступ: 2025-02-11.

[65]. Acm digital library. https://dl.acm.org. Доступ: 2025-02-11.

[66]. Springerlink. https://link.springer.com. Доступ: 2025-02-11.

[67]. Github. https://github.com. Доступ: 2025-02-11.

[68]. Goroutines. https://go.dev/doc/effective_go#goroutines. Доступ: 2025-02-11.

[69]. Go channels. https://go.dev/doc/effective_go#channels. Доступ: 2025-02-11.

[70]. C. Hoare. Communicating sequential processes. В Theories of Programming: The Life and Works of Tony Hoare, страницы 157–186. 2021.

[71]. T. Tu, X. Liu, L. Song и Y. Zhang. Understanding real-world concurrency bugs in go. В Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems, страницы 865–878, 2019.

[72]. K. Honda, N. Yoshida и M. Carbone. Multiparty asynchronous session types. В Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, страницы 273–284, 2008.

[73]. Migoinfer+. https://github.com/JujuYuki/gospal. Доступ: 2025-02-11.

[74]. Godel 2 benchmarks. https://github.com/JujuYuki/godel2-benchmark. Доступ: 2025-02-11.

[75]. O. Bunte, J. F. Groote, J. J. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs и T. A. Willemse. The mcrl2 toolset for analysing concurrent systems: improvements in expressivity and usability. В Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II 25, страницы 21–39. Springer, 2019.

[76]. Promela. https://en.wikipedia.org/wiki/Promela. Доступ: 2025-02-11.

[77]. Spin. https://spinroot.com/spin/whatispin.html. Доступ: 2025-02-11.

[78]. Gomela. https://github.com/nicolasdilley/gomela-ase21/. Доступ: 2025-02-11.

[79]. Gcatch. https://github.com/system-pclub/GCatch. Доступ: 2023-10-05.

[80]. Crypto. https://pkg.go.dev/crypto. Доступ: 2025-02-11.

[81]. Crypto. https://pkg.go.dev/golang.org/x/crypto. Доступ: 2025-02-11.

[82]. Go-safer. https://github.com/jlauinger/go-safer. Доступ: 2025-01-15.

[83]. Goanalysis. https://github.com/chrisbbe/GoAnalysis. Доступ: 2025-02-11.

[84]. T. J. McCabe. A complexity measure. IEEE Transactions on software Engineering, (4):308–320, 1976.

[85]. S. M. Olbrich, D. S. Cruzes и D. I. Sjøberg. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. В 2010 IEEE international conference on software maintenance, страницы 1–10. IEEE, 2010.

[86]. R. Harrison, S. J. Counsell и R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. IEEE Transactions on Software Engineering, 24(6):491–496, 2002.

[87]. R. Subramanyam и M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Transactions on software engineering, 29(4):297–310, 2003.

[88]. P. Anderson, D. Binkley, G. Rosay и T. Teitelbaum. Flow insensitive points-to sets. Information and Software Technology, 44(13):743–754, 2002. DOI: https://doi.org/10.1016/S0950-5849(02)00105-2. URL: https://www.sciencedirect.com/science/article/pii/S0950584902001052. Special Issue on Source Code Analysis and Manipulation (SCAM).

[89]. Coverity 2021.03: Supported Platforms. Доступ: 2025-02-11. 2021. URL: https://sigdocs.synopsys.com/polaris/topics/r_coveritycompatible-platforms_2021.03.html.

[90]. Common weakness enumeration. https://cwe.mitre.org. Доступ: 2024-10-01.

[91]. Badgerdb. https://github.com/hypermodeinc/badger. Доступ: 2025-02-11.

[92]. J. Hu, L. Zhang, C. Liu, S. Yang, S. Huang и Y. Liu. Empirical analysis of vulnerabilities life cycle in Golang ecosystem. В Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, страницы 1–13, 2024.

[93]. About - git. https://git-scm.com/about/data-assurance. Доступ: 2025-02-11.

[94]. Go modules services. https://proxy.golang.org/. Доступ: 2025-02-11.

[95]. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman и F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4):451–490, окт. 1991. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[96]. Ssadump. https://pkg.go.dev/golang.org/x/tools/cmd/ssadump. Доступ: 2023-10-05.

[97]. C. Lattner и V. Adve. A compilation framework for lifelong program analysis and transformation. В CGO, том 4, страница 75, 2003.

[98]. Gollvm is an llvm-based Go compiler. https://go.googlesource.com/gollvm/. Доступ: 2024-10-05.

[99]. N. Malyshev, I. Dudina, D. Kutz, A. Novikov и S. Vartanov. Smt solvers in application to static and dynamic symbolic execution: a case study. В 2019 Ivannikov Ispras Open Conference (ISPRAS), страницы 9–15. IEEE, 2019.

[100]. W. Landi. Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS), 1(4):323–337, 1992.

Информация об авторах / Information about authors

Варвара Викторовна ДВОРЦОВА – сотрудник ИСП РАН, аспирант ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, анализ Golang.

Varvara Viktorovna DVORTSOVA – ISP RAS researcher, postgraduate student at ISP RAS. Her research interests: compiler technologies, static analysis, Golang analysis.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), senior researcher. His research interests: static analysis for finding errors in source code.

DOI: 10.15514/ISPRAS-2025-37(6)-5

Повышение точности статического анализа кода при помощи больших языковых моделей

Д.Д. Панов, ORCID: 0009-0003-5809-1187 <d.panov@ispras.ru>
Н.В. Шимчик, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
Д.А. Чибисов, ORCID: 0009-0002-1198-3148 <dchibisov@ispras.ru>
А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>
В.Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>

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

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

Аннотация. В данной работе описывается подход к проверке результатов статического анализа кода при помощи больших языковых моделей (LLM), выполняющий фильтрацию предупреждений с целью удаления ложных. Для составления запроса к LLM предложенный подход сохраняет информацию, собранную анализатором, такую как абстрактное синтаксическое дерево программы, таблицы символов, резюме типов и функций. Эта информация может как напрямую передаваться в запросе к модели, так и использоваться для более точного определения фрагментов кода, необходимых для проверки истинности предупреждения. Подход был реализован в SharpChecker – промышленном статическом анализаторе для языка C#. Его тестирование на реальном коде показало повышение точности результатов на величину до 10 процентных пунктов при сохранении высокой полноты (от 0,8 до 0,97) для чувствительных к контексту и путям межпроцедурных детекторов утечки ресурсов, разыменования null и целочисленного переполнения. Для детектора недостижимого кода применение информации из статического анализатора позволило повысить полноту на 11–27 процентных пунктов по сравнению с подходом, использующим в запросе только исходный код программы.

Ключевые слова: статический анализ кода; большие языковые модели LLM.

Для цитирования: Панов Д.Д., Шимчик Н.В., Чибисов Д.А., Белеванцев А.А., Игнатьев В.Н. Повышение точности статического анализа кода при помощи больших языковых моделей. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 83–100. DOI: 10.15514/ISPRAS–2025–37(6)–5.

Increasing Precision of Static Code Analysis Using Large Language Models

D.D. Panov, ORCID: 0009-0003-5809-1187 <d.panov@ispras.ru>
N.V. Shimchik, ORCID: 0000-0001-9887-8863 <shimnik@ispras.ru>
D.A. Chibisov, ORCID: 0009-0002-1198-3148 <dchibisov@ispras.ru>
A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>
V.N. Ignatyev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>

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

Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. This paper describes an approach to verifying the results of static code analysis using large language models (LLMs), which filters warnings to eliminate false positives. To construct the prompt for LLM, the proposed approach retains information collected by the analyzer, such as abstract syntax trees of files, symbol tables, type and function summaries. This information can either be directly included in the prompt or used to accurately identify the code fragments required to verify the warning. The approach was implemented in SharpChecker – an industrial static analyzer for the C# language. Testing on real-world code demonstrated an improvement in result precision by up to 10 percentage points while maintaining high recall (0.8 to 0.97) for context-sensitive and interprocedural path-sensitive detectors of resource leaks, null dereferences, and integer overflows. In case of unreachable code detector, use of information from the static analyzer improved recall by 11–27 percentage points compared to an approach that only uses the program's source code in the prompt.

Keywords: static code analysis; large language models LLM.

For citation: Panov D.D., Shimchik N.V., Chibisov D.A., Belevantsev A.A., Ignatyev V.N. Increasing precision of static code analysis using large language models. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 83-100 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-5.

1. Введение

Статический анализ исходного кода является важным подходом к поиску ошибок и уязвимостей и позволяет искать ошибки в коде программы без ее запуска. Одной из проблем, возникающих в ходе применения статических анализаторов, является большое количество предупреждений, выдаваемых на больших проектах. Средняя плотность предупреждений промышленного статического анализатора составляет 10 сообщений на 1000 строк кода, что даже при высокой точности, достигающей 90% истинных предупреждений, приводит к сотням ложных срабатываний. Более сложные детекторы ошибок, как например, поиск утечек ресурсов, разыменований null могут выдавать до 40% ложных предупреждений. На их анализ тратятся ресурсы квалифицированных разработчиков. Поэтому актуальна задача автоматизации разметки или фильтрации ложных предупреждений. Ее основная сложность состоит в исключении только ложных предупреждений без потери истинных.

В современном промышленном анализаторе очень сложно добиться повышения точности без снижения полноты только за счет усовершенствования имеющихся алгоритмов и моделей программы. Поэтому для фильтрации ложных предупреждений можно использовать *верификацию* – их дополнительную проверку другими алгоритмами. Например, предупреждения нечувствительного к путям анализа помеченных данных можно проверять динамическим анализом или символьным выполнением [1], результаты чувствительного к путям анализа – методами машинного обучения [2].

В настоящее время одними из самых совершенных методов на основе машинного обучения является использование больших языковых моделей (Large Language Model, LLM). Тривиальным подходом их применения к задаче верификации является запрос к LLM на

основе предупреждения анализатора и небольшой окрестности кода (десятки строк) вокруг. Это дает неплохие результаты, но только для тех случаев, когда весь код, важный для оценки истинности предупреждения, попадает в запрос [3]. Это условие на реальных проектах выполняется в основном для детекторов, анализирующих абстрактное синтаксическое дерево (АСД), а для межпроцедурных случаев или ситуаций, требующих изучения нескольких фрагментов кода, не дает существенных результатов, поскольку его истинность зависит от контекста, находящегося за пределами текущей функции или даже файла. Поскольку размер контекстного окна в больших языковых моделях ограничен, а также качество ответа модели ухудшается с ростом размера запроса, важной задачей является поиск релевантной для понимания текущего предупреждения информации.

Применение больших языковых моделей для верификации в промышленном статическом анализаторе сопряжено с дополнительными ограничениями. Так, для защиты пользовательского кода, часто составляющего коммерческую тайну, от доступа третьих лиц исключена возможность использования больших моделей, предоставляемых в виде сервиса через интернет, как например, GPT-4 [4]. А для локального развертывания доступны лишь модели ограниченного размера с открытыми весами. Однако преимуществом использования таких моделей является возможность их дообучения на корпусе размеченных предупреждений анализатора. Кроме того, подходы, демонстрирующие высокие показатели на синтетических тестах с ошибками, никогда не достигают аналогичных показателей на реальном коде.

В данной работе предлагается метод верификации предупреждений промышленного статического анализатора Svace на основе специально собранной в процессе анализа информации. Собранная информация включает резюме методов и типов, содержащие их основные, важные для понимания ошибки свойства; АСД для извлечения минимальных осмысленных блоков кода, связанных с каждой точкой трассы предупреждения и другие данные. Кроме этого, для повышения точности и полноты фильтрации применяется дообучение на наборе из сотен истинных и ложных предупреждений на реальных проектах, собранных и размеченных в процессе разработки анализатора. Предложенный метод реализован для языка C# в анализаторе SharpChecker [5] (является частью Svace [6]). Реализация подхода для сложных детекторов ошибок разыменования null, утечки ресурсов и целочисленного переполнения, найденных чувствительным к путям межпроцедурным анализом в коде на языке C#, повышает их совокупную точность на 11 процентных пунктов до 86% при полноте 91% на реальных проектах. Для детектора недостижимого кода полнота фильтрации предупреждений не достигла 70%, но был заметен её прирост благодаря использованию информации, собранной статическим анализатором.

Структура данной статьи следующая. О существующих подходах к верификации результатов статического анализатора с помощью методов машинного обучения рассказывается в разделе 2. Общая схема предлагаемого в данной работе метода описывается в разделе 3. Сбор информации из статического анализатора описывается в разделе 4. Генерация запросов и интерпретация ответов модели описывается в разделе 5. Результаты тестирования полученного решения при анализе проектов с открытым исходным кодом проводится в разделе 6.

2. Обзор существующих решений

Существуют различные подходы к подтверждению результатов статического анализа, начиная с воспроизведения трассы предупреждения при помощи динамического анализа или фаззинга, заканчивая подходами на основе машинного обучения, которые предназначены для оценки вероятности истинности предупреждения.

Остановимся подробнее на подходах, использующих большие языковые модели.

В работе Li H. et al. [7] был представлен инструмент LLift, объединяющий статический анализатор UBITech и большие языковые модели. Схема работы инструмента состояла из трёх этапов:

1) нечувствительное к путям символьное выполнение, позволяющее определить множество возможных ошибок;
2) чувствительный к путям анализ, после которого часть ошибок подтверждается или отсеивается, а часть остаётся неклассифицированными;
3) для неклассифицированных потенциальных ошибок создаётся запрос к большой языковой модели (в качестве примера – GPT-4), включающий в себя собранную информацию об инициализаторах всех имеющих отношение к предупреждению переменных.

Данный подход смог продемонстрировать 50% точность при анализе ядра Linux без потерянных истинных срабатываний и показал пользу от использования вычисляемой анализатором информации в запросах к моделям.

В работе Mohajer M. et al. [8] описывается подход к использованию больших языковых моделей для проверки предупреждений типов «разыменование нулевого указателя» и «утечка ресурсов». Авторам удалось достичь повышения точности результатов статического анализатора Infer с 65% до 94% и с 54% до 63% соответственно, однако продемонстрированная полнота оказалась заметно ниже – 64% и 80% соответственно.

В работе Wen C. et al. [9] описывается инструмент LLM4SA, также решающий задачу проверки истинности предупреждений с помощью больших языковых моделей. Авторы предложили решать проблему извлечения релевантного контекста из исходного кода при помощи графа зависимостей программы, отображающего зависимости по данным и по управлению между инструкциями. Демонстрируемый модели фрагмент кода включает в себя не только ближайшую окрестность, но и те части исходного кода, от которых может зависеть код, в котором было выдано предупреждение. Также для повышения стабильности результатов авторы выполняли больше одного запроса к модели и усредняли результаты классификации.

Если для какого-то предупреждения не было заметного перевеса в сторону «истинности» или «ложности», оно помечалось как «неопределённое». Помимо GPT-3.5-Turbo, авторы продемонстрировали часть результатов с использованием языковой модели с открытыми весами Llama-2-70B и сделали вывод о том, что та смогла продемонстрировать сравнимые результаты. При тестировании на реальных проектах инструмент показал полноту в 93%, однако общую точность около 5%, что в первую очередь вызвано крайне низкой точностью выбранных авторами статических анализаторов.

В работе Li Z. et al. [10] описывается инструмент IRIS, в котором большие языковые модели используются для улучшения результатов статического анализа помеченных данных, проводимого инструментом CodeQL. Использование языковых моделей для пополнения списка используемых инструментом истоков, стоков и пропагаторов помеченных данных позволило более чем в два раза повысить количество обнаруженных уязвимостей. Для повышения точности они использовали запросы, в которых предоставляли модели контекст найденных предупреждений и просили дать текстовый комментарий для демонстрации пользователю, а также оценить, является ли предупреждение истинным. Использование GPT-4 позволило поднять нижнюю оценку точности с 10% до 15%, в то время как для других использованных в работе моделей повышение полноты анализа сопровождалось падением его точности. После ручной проверки выборки из 50 предупреждений авторы получили оценку точности, равную 54% при использовании GPT-4.

В работе Khare A. et al. [11] проводится общая оценка применимости 16 больших языковых моделей к поиску уязвимостей на наборе из 5000 тестов, частично состоящих из

искусственных примеров и частично основанных на коде реальных проектов. Хотя эта работа напрямую не связана с верификацией результатов статического анализа, авторы делают ряд полезных выводов, в частности: отдельные сравнительно небольшие модели (14 и 32 миллиарда параметров) смогли показать лучшие результаты на реальных проектах, чем значительно бóльшие модели, включая GPT-4. Также авторы отметили некоторые типы уязвимостей, для которых языковые модели показали лучший результат, чем статический анализатор CodeQL – в основном это предупреждения, для оценки которых не требуется знание глобального контекста или понимание сложных структур данных.

Таким образом, хотя использование больших языковых моделей для фильтрации предупреждений статического анализатора активно исследуется, среди рассмотренных нами работ не нашлось таких, которые бы продемонстрировали одновременно высокую точность и высокую полноту результатов. Кроме того, эти работы в первую очередь ориентировались на проприетарные языковые модели, которые отличаются от открытых большим количеством весов, но невозможностью их локального запуска на собственном оборудовании, необходимостью передачи демонстрируемого в запросе исходного кода третьим лицам, а также ограниченными возможностями по дообучению модели под свои нужды.

3. Схема разработанного метода

При ответе на запрос о верификации предупреждения LLM генерирует результат как на основе информации, представленной в запросе, так и использует закодированные в модели данные, например, информацию о популярных библиотечных функциях или даже исходный код анализируемого открытого проекта, если он был использован при обучении модели. Метод, предложенный в работе, основан на предположении, что любая информация, присутствующая в коде или доступная анализатору, которая может влиять на истинность предупреждения, должна быть включена в запрос к модели. В противном случае истинность предупреждения невозможно установить корректно даже аналитику. Таким образом, для каждого предупреждения требуется выделить необходимую для его проверки информацию:

1) фрагменты кода, влияющие на истинность срабатывания, например условия ветвлений;
2) функции, переменные, типы и их свойства, например, инициализация неизменяемых (readonly) переменных или аллокацию ресурсов.

Тривиальное вырезание фрагментов кода из нескольких строк до и после вставляет в запрос обрывки операторов, которые могут усложнить его понимание, поэтому требуется выделение структурных блоков. А для получения свойств функций, например, при каких условиях она может выбросить исключение, требуется нетривиальный анализ. К счастью, результаты такого анализа доступны во время работы статического анализатора и необходимо лишь их упрощение и экспорт в понятном для модели формате. Например, анализатор задает условия в терминах идентификаторов символьных значений, а модель сможет их воспринять лишь в терминах переменных, определенных в коде.

Во многих статических анализаторах предупреждение описывается не одной точкой в программе и текстовым описанием ошибки, а целой последовательностью точек, помогающих человеку лучше понять детали ошибки. Эта последовательность пар «точка в программе, текстовое описание» называется *трассой предупреждения*. Например, для ошибки «разыменование null» анализатор может показать условия, при которых переменная может стать null, а также путь выполнения, достигающий точки разыменования. Таким образом, наличие трассы упрощает поиск релевантных фрагментов кода. Однако для выявленных локаций в коде остается задача выделения фрагмента кода и встраивания сообщений трассы в запрос. Например, их можно вставлять в код в виде комментария в конце строки или строкой выше, или перечислять отдельно с указанием номеров строк.

Включение в запрос исходного кода всех связанных функций невозможно, т.к. они в свою очередь могут зависеть от других функций и типов. Поэтому необходимо исследование различных способов добавления их резюме в запрос.

После выполнения запросов каждый ответ модели должен быть классифицирован как истинное или ложное предупреждение, после чего в итоговом отчёте каждому исходному предупреждению присваивается соответствующий статус, а также опционально – указывается текст ответа модели. Интерпретация ответов модели также представляет исследовательский интерес, поскольку модель генерирует ответы в виде текста в свободной форме, а их классификация должна выполняться программно без участия человека – подробнее о способах решения этой проблемы рассказывается в разделе 5.

Таким образом, метод верификации состоит из трех рассмотренных далее этапов, представленных на рисунке 1: экспорт информации из анализатора; построение набора запросов к модели на основе предупреждений и сохраненной информации и интерпретация ответов модели. Построенная схема также используется и для дообучения на размеченных предупреждениях.

Рис. 1. Общая архитектура решения.
Fig. 1. General structure of the solution.

4. Сбор и сохранение информации о программе

В данном разделе рассматриваются структуры данных, которые получены из статического анализатора и содержат информацию для понимания контекста предупреждения. Для ее извлечения в анализаторе SharpChecker был реализован отдельный модуль, сохраняющий всю необходимую информацию в виде файлов в формате JSON, которые будут далее использованы при генерации запросов к LLM.

Предупреждение анализатора включает основную точку в коде, сообщение, *трассу*, состоящую из пар [точка в коде, сообщение], а также некоторую служебную информацию, как например полное имя метода, содержащего предупреждение. В анализаторе SharpChecker для хранения предупреждений используется XML-подобный формат Svace, подходящий для дальнейшего использования, поэтому ничего дополнительно сохранять не требуется.

Абстрактное синтаксическое дерево (АСД) применяется в компиляторах на этапе синтаксического анализа для описания иерархической структуры кода. Для получения АСД не обязательно встраиваться в анализатор, а возможно использование сторонних, мульти-языковых инструментов, как например, tree-sitter [12]. Однако они не всегда способны дать точный результат из-за возможных неточностей в грамматиках, отсутствия поддержки новых стандартов языка или отсутствия информации о содержимом подключаемых заголовочных файлов. Экспорт АСД непосредственно из компилятора или анализатора позволяет учитывать все особенности сборки: макроопределения, заголовочные и генерируемые файлы и т.д.

Существуют подходы [13] на основе машинного обучения, в которых АСД являются частью входной информации модели, однако большие языковые модели предназначены для работы с текстовой информацией, потому в данном подходе синтаксические деревья играют вспомогательную роль, предоставляя информацию о границах функций и отдельных блоков кода, для выделения окрестности кода – например, для многострочных операторов.

Статический анализатор SharpChecker основан на компиляторной инфраструктуре Roslyn [14] и использует ее представление АСД. Поскольку в настоящее время АСД используется только для выделения осмысленного законченного фрагмента кода для произвольной точки в программе, сохраняется лишь необходимая его часть: вершины объявлений пространств имён, типов, методов, полей, свойств, всех видов функций Перечисленные вершины, начиная с корня дерева, преобразуются в JSON формат с атрибутами, задающими участок кода, соответствующий вершине; тип вершины; имя объявляемой сущности и множество непосредственных потомков.

Таблица символов позволяет сопоставить идентификаторам в текущей области видимости соответствующие им объявления переменных, типов, функций. В анализаторе Svace таблица символов доступна как из компилятора (Roslyn для C#), так и из собранной для навигации по коду информации базы данных в формате DXR [15]. Преимущество DXR состоит в единообразном представлении для всех поддерживаемых анализатором языков программирования. Помимо поиска объявлений, в ней содержится информация из графа наследования, а также необходимые структуры для поиска всех использований символа и переопределений виртуальных методов.

Резюме типа содержит важные для анализа свойства типа данных. Примерами таких свойств являются имя, разновидность (примитивный тип, структура, класс, интерфейс и т.д.), родители, наследники, список полей класса, а также различные атрибуты типа.

Резюме типов можно подставлять в текст запроса вместо определений в форме кода. Они имеют унифицированный формат и обычно являются более компактными, чем определения соответствующих типов в исходном коде. Исходные данные для экспорта вычисляются анализатором SharpChecker и включают как общую информацию, например, является ли тип абстрактным, так и специальную, например, о том, что тип является ресурсом (реализует интерфейс IDisposable).

Резюме функций в статических анализаторах обычно используется для обеспечения масштабируемости анализа за счёт предварительного однократного вычисления ключевых свойств функции и их применения в точке вызова. Содержимое резюме функции зависит от поддерживаемых анализатором типов ошибок. Например, для детектора «разыменование null» существенной информацией является то, может ли функция вернуть значение null, а также то, при каких условиях в ней происходят разыменования аргументов.

Резюме экспортируется после анализа каждой функции и содержит имя и сигнатуру функции; ее тип (обычный метод, конструктор, аксессор, делегат и т. д.); возвращаемое значение; флаг, определяющий, может ли функция вернуть null; список измененных полей вместе с их новыми значениями; список выбрасываемых исключений с соответствующим условием. Помимо вышеперечисленных атрибутов, каждый детектор может помещать в резюме специализированные свойства функции, необходимые для поиска конкретного типа ошибок. К примеру, детектор ошибок типа разыменование значения null записывает в резюме массив условий возможного разыменования значений в данной функции; детектор ошибок типа утечка ресурсов записывает в резюме информацию о том, была ли вызвана функция закрытия ресурса, возвращает ли функция новый ресурс или уже существующий, какие ресурсы закрываются в функции и какие ресурсы сохраняются в объекте.

Использование анализатора в качестве источника данных позволяет сохранить в резюме предикаты (условия) различных событий, таких как выброс исключения, освобождение ресурса. Эти предикаты строятся при статическом символьном выполнении и представляют

собой формулы и выражения над символьными переменными. Для включения в запрос требуется их трансляция в формулу над переменными, объявленными в коде программы. Кроме того, полученные в результате символьного выполнения предикаты могут быть сложными формулами, которые все равно не будут корректно интерпретированы современными LLM, поэтому такие выражения либо заменяются на неизвестное условие, либо упрощаются. Пример трансляции внутреннего представления значений в читаемый вид приведен на рис. 2.

```
{
    "variable": "m3735",
    "condition": "[!pure3729 && !pure3732]"
}
{
    "variable": "_salt",
    "condition": "[!(string.IsNullOrEmpty(plainText)) and !(string.
       IsNullOrEmpty(sharedSecret))]"
}
```

Рис. 2. Пример преобразования символьных выражений в формулу над переменными, объявленными в коде.

Fig. 2. An example of transforming symbolic expressions into a formula over variables declared in the code.

При анализе в SharpChecker переменной _salt было сопоставлено символьное значение m3735, а условие ее разыменования построено в виде логической формулы над возвращаемыми значениями вызовов метода string.IsNullOrEmpty с различными аргументами.

Таким образом, использование резюме методов позволяет подать на вход модели в удобном формате важную для проверки истинности предупреждения информацию, использованную анализатором для его обнаружения. Достоверность такой информации высока, но не абсолютна, что может послужить причиной ошибки при верификации.

5. Генерация и интерпретация запроса

Модуль генерации в качестве входных данных принимает исходный код анализируемого проекта, результаты статического анализатора и базу данных с экспортированной информацией.

На первом этапе выполняется загрузка информации. Анализируемый проект может содержать сотни тысяч методов и десятки тысяч предупреждений, поэтому важно обеспечить эффективный менеджмент памяти, загружая только необходимые данные.

Генерация запроса выполняется независимо для каждого предупреждения. Сначала выделяется контекст в окрестности места предупреждения в коде. Контекст может задаваться количеством строк до и после, либо выбирается функция целиком. Соответствующие контексту строки помечаются.

Точки трассы представляются в виде комментариев в коде. При этом строки вокруг них также помечаются. В случае агентного подхода, возможна самостоятельная навигация модели по коду вдоль трассы предупреждения вместо предоставления ей фрагментов кода вокруг всех межпроцедурных точек трассы в одном запросе.

Далее формируется множество символов, которые используются внутри выделенных фрагментов кода. При помощи экспортированной таблицы символов в каждом участке кода происходит поиск всех идентификаторов и их свойств, таких как полное имя, вид (переменная, функция или тип), ссылка на определение и список ссылок на все использования. В запрос можно добавлять как определения переменных и типов в виде кода,

так и отдельные их свойства. Поведение определяется выбранной стратегией извлечения символов:

- без извлечения символов;
- с извлечением символов внутри участка кода вокруг предупреждения;
- с извлечением символов внутри участка кода не только вокруг предупреждения, но и вокруг точек трассы;
- с рекурсивным извлечением новых символов их определений уже добавленных.

При агентном подходе модели предоставлена возможность самостоятельно запрашивать информацию об идентификаторах, фигурирующих в исходном коде. Для каждой переменной из сформированного множества символов выделяются строки с её определением.

При использовании резюме функций и типов, формируется множество релевантных резюме. Оно определяется ранее построенным множеством релевантных символов или, если символы не используются, информацией о функциях, вызываемых из текущей. Основным форматом для представления резюме является JSON, как изображено на рис. 3. Также был опробован подход с его переводом в текстовое описание, но он не оказал существенного влияния на результаты модели.

```
Summary for Ordinary method ‘Pop‘:
{
    "Prototype": "NetMQ.NetMQFrame NetMQ.NetMQMessage.Pop()",
    "CanReturnNull": True,
    "ReturnValue": "this.m_frames.Object(0)",
    "ChangedVariables": [],
    "DereferenceConditions": [
        {"variable": "this.m_frames", "condition": "true"}
    ]
}
```

Рис. 3. Пример резюме для метода.
Fig. 3. Example of method summary.

Поля резюме можно фильтровать по именам полей, типам предупреждений и неизвестным предикатам.

Наконец, для формирования запроса требуется объединить релевантные предупреждению строки кода в один фрагмент для демонстрации модели. В простейшем случае берётся основной контекст в виде заданного количества строк выше и ниже предупреждения без выхода за границы текущей функции. К основному контексту могут быть добавлены помеченные строки вокруг точек трассы и с определениями переменных. Если используется АСД, то помимо этого осуществляется проход от всех вершин, в которых были помечены строки, к корню дерева: для листовых вершин соответствующие строки кода добавляются в итоговый фрагмент кода целиком, в то время как для внутренних вершин добавляется их заголовок (например, сигнатура функции или условие цикла) без тела оператора. Множество полученных строк объединяется в фрагменты кода с разбиением по файлам.

Помимо построения запроса исследовательской задачей является и его интерпретация. Для однозначной классификации ответов модели рассмотрено несколько подходов.

1) В запросе можно потребовать начать ответ с «Да» или «Нет», после чего проверять наличие этих слов в начале строки. В редких случаях модель игнорирует это требование. Также такой подход требует адаптации при работе с «рассуждающими» моделями, поскольку те сначала приводят последовательность размышлений и только потом дают окончательный ответ – в этом случае обработчику нужно знать ключевые слова, с которых начинается секция ответа. Во многих моделях для

упрощения последующей обработки это оформляется в виде одного или двух xml тегов, например: “<think>...</think> <answer>Ответ</answer>”.

2) Модели разрешается давать ответ в свободной форме, но после выполняется дополнительный запрос, который требует свести предыдущий ответ к простому «Да» или «Нет». Такой подход наиболее универсален, но влияет на общее время работы из-за выполнения дополнительных запросов.
3) Большинство современных систем взаимодействия с LLM поддерживает ограничение вывода модели при помощи JSON схем, при котором ответ выводится в виде корректного JSON объекта с заданным набором полей. Ключевым недостатком является то, что такой подход не подходит для рассуждающих моделей, так как не предоставляет пространства для размышлений перед окончательным ответом.
4) Некоторые системы поддерживают ограничение вывода модели при помощи контекстно-свободных грамматик. Они позволяют контролировать вывод модели с произвольной строгостью и могут заменить любой из перечисленных выше подходов, благодаря чему ответы становится легко классифицировать. Из возможных побочных эффектов – качество результатов может снижаться, если грамматика сильно ограничивает множество возможных ответов, а сам формат ответов существенно отличается от того, как модель ответила бы без применения грамматики.

В данной работе тестирование проводилось с использованием первого подхода. Отдельные эксперименты, проводимые другими методами интерпретации, показали, что использование информации анализатора сопоставимо влияет на результаты во всех случаях.

6. Тестирование

Тестирование подхода проводилось с использованием набора предупреждений статического анализатора SharpChecker, являющегося частью Svace. Они были получены 4 детекторами различных типов при анализе 15 проектов с открытым исходным кодом на языке C#, таких как Roslyn, Lucene.NET, OpenSimulator и других, использующихся для тестирования в ходе разработки инструмента, а потому имеющих достаточное количество размеченных вручную предупреждений.

Для тестирования были выбраны следующие большие языковые модели:

1) Phind/Phind-CodeLlama-34B-v2 – версия модели CodeLlama, специализирующаяся на решении задач, связанных с кодом, сокращённо будем называть её Phind;
2) Qwen/Qwen2.5-Coder-32B-Instruct – модель схожего размера, также специализирующаяся на работе с кодом;
3) deepseek-ai/DeepSeek-R1-Distill-Qwen-32B – «рассуждающая» модель, полученная путём переобучения модели Qwen2.5-32B на выводе модели DeepSeek-R1, сокращённо будем называть её R1-Qwen;
4) deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B – аналог предыдущей модели, но значительно меньшего размера, который можно применять даже без видеокарты;
5) deepseek-ai/DeepSeek-R1-Distill-Llama-70B – аналог предыдущей модели, но с 70 миллиардами параметров и использующий в качестве базовой модели Llama-3.3.

Для детерминизма результатов, все запуски выполнялись с температурой 0. Выполнение запросов осуществлялось с использованием библиотеки vLLM [16] на Nvidia A100 с 80 Гб видеопамяти. Для запуска 70B модели потребовалось использовать две такие видеокарты.

Основными метриками оценки результатов являются точность, полнота и F_1-мера. Точность показывает процент истинных срабатываний среди предупреждений, оставшихся после

фильтрации, и считается по формуле $P = \frac{TP}{TP+FP}$. Под полнотой R понимается отношение числа истинных срабатываний, оставшихся после фильтрации, к изначальному количеству истинных предупреждений, найденных статическим анализатором, и считается по формуле$R = \frac{TP}{TP+FN}$ – таким образом, полнота результатов до фильтрации принимается за единицу. F_1-мера считается по формуле $F_1 = 2\frac{P \cdot R}{P+R} = \frac{2TP}{2TP+FP+FN}$.

Текстовый запрос к модели имеет следующую структуру:

- указание на язык программирования;
- исходный код, выделенный для предупреждения описанным выше методом, обрамлённый символами ``;
- резюме типов и методов в формате json;
- описание типа предупреждения и его сообщения;
- вопрос об истинности предупреждения с указанием желаемого формата ответа и критериями для вынесения вердикта: ответ должен быть пошагово объяснён внутри тега “<think>”, окончательный ответ должен состоять из “Yes” или “No” в теге “<output>”, ответ “No” следует давать только в случае уверенности в ложности данного предупреждения.

В табл. 1, 2, 3 и 4 приведены подробные результаты фильтрации предупреждений типов «утечка ресурсов», «разыменование null», «недостижимый код» и «целочисленное переполнение» соответственно при использовании двух моделей: Phind-CodeLlama-34B-v2 и DeepSeek-R1-Distill-Qwen-32B. В этих таблицах:

1) столбец «АСД» показывает, использовались ли при построении запроса абстрактные синтаксические деревья для более точного определения границ функций и многострочных операторов;
2) столбец «Символы» показывает стратегию добавления в запрос строк кода, содержащих релевантных символы: без дополнительных строк, с добавлением кода в окрестностях точек трассы предупреждения и с рекурсивным добавлением символов, фигурирующих в определениях других символов;
3) столбец «Резюме» показывает, вставляются ли в запрос резюме функций и типов;
4) столбец «W_a; W_b» обозначает размер основного контекста – количество строк выше и ниже предупреждения, включаемых в запрос;
5) столбец «UC» (unclear) показывает количество запросов к модели, ответы на которые не удалось получить или классифицировать, а потому они не были отнесены ни к одной из четырёх категорий истинности/положительности предупреждения.

Отметим, что для детектора недостижимого кода не удалось достичь приемлемой полноты фильтрации, из-за чего F_1 мера оказалась значительно ниже исходного показателя. Тем не менее, полнота результатов при применении предложенного подхода оказалась на 11–27 процентных пунктов выше, чем без использования информации, предоставляемой статическим анализатором.

Для детектора целочисленного переполнения при использовании рассуждающей модели на любой конфигурации запроса точность фильтрации повысилась на 16–20 процентных пунктов (до 0,73–0,77) относительно базовой точности статического анализатора при полноте около 0,81. При этом использование информации, предоставляемой статическим анализатором, не показало значительного прироста к точности или полноте по сравнению с подходом, что может быть связано с отсутствием в резюме информации, важной для понимания предупреждений для данного детектора, а также тем, что при определении

релевантных фрагментов кода в них не включаются использования задействованных переменных.

Табл. 1. Результаты для детектора утечки ресурсов.
Table 1. Results for resource leak checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				374	0	54	0	0	0,87	1	0,93
Phind	-	-	-	300; 2	364	2	52	10	0	0,88	0,97	0,92
	-	-	+	300; 2	260	7	46	114	1	0,85	0,7	0,76
	+	-	-	2; 2	340	5	49	34	0	0,87	0,91	0,89
	+	по трассе	-	2; 2	330	7	47	44	0	0,88	0,88	0,88
	+	по трассе	+	2; 2	362	4	50	12	0	0,88	0,97	0,92
	+	рекурсивно	+	0; 0	366	5	49	8	0	**0,88**	**0,98**	0,93
R1-Qwen	-	-	-	300; 2	333	17	37	41	0	0,9	0,89	0,9
	-	-	+	300; 2	362	9	45	12	0	0,89	0,97	0,93
	+	-	-	2; 2	342	18	36	32	0	0,9	0,91	0,91
	+	по трассе	-	2; 2	340	17	37	34	0	0,9	0,91	0,91
	+	по трассе	+	2; 2	360	20	34	13	1	**0,91**	**0,97**	**0,94**
	+	рекурсивно	+	0; 0	358	10	43	16	1	0,89	0,96	0,92

Табл. 2. Результаты для детектора разыменования null.
Table 2. Results for null dereference checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				331	0	115	0	0	0,74	1	0,85
Phind	-	-	-	300; 2	325	3	112	6	0	0,74	0,98	0,85
	-	-	+	300; 2	262	42	73	68	1	**0,78**	0,79	0,79
	+	-	-	2; 2	311	9	106	20	0	0,75	0,94	0,83
	+	по трассе	-	2; 2	306	14	101	25	0	0,75	0,92	0,83
	+	по трассе	+	2; 2	218	57	57	112	2	**0,79**	0,66	0,72
	+	рекурсивно	+	40; 2	263	28	86	68	1	0,75	0,79	0,77
R1-Qwen	-	-	-	300; 2	266	75	39	64	2	0,87	0,81	0,84
	-	-	+	300; 2	256	50	57	55	28	0,82	0,82	0,82
	+	-	-	2; 2	268	69	45	63	1	0,86	0,81	0,83
	+	по трассе	-	2; 2	280	76	39	46	5	**0,88**	**0,86**	**0,87**
	+	по трассе	+	2; 2	295	61	54	35	1	**0,85**	**0,89**	**0,87**
	+	рекурсивно	+	40; 2	284	64	49	44	5	0,85	0,87	0,86

Табл. 3. Результаты для детектора недостижимого кода.
Table 3. Results for unreachable code checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				445	0	184	0	0	0,71	1	0,83
Phind	-	-	-	300; 2	185	109	75	260	0	0,71	0,42	0,52
	-	-	+	300; 2	280	57	126	163	3	0,69	0,64	0,66
	+	-	-	2; 2	263	44	137	182	3	0,66	0,59	0,62
	+	по трассе	-	2; 2	272	57	123	172	5	0,69	0,61	0,65
	+	по трассе	+	2; 2	307	53	128	138	3	0,71	**0,69**	0,7
R1-Qwen	-	-	-	300; 2	217	104	63	210	35	0,78	0,51	0,61
	-	-	+	300; 2	206	86	74	225	38	0,74	0,48	0,58
	+	-	-	2; 2	234	98	75	201	21	0,76	0,54	0,63
	+	по трассе	-	2; 2	250	96	72	182	29	0,78	0,58	0,66
	+	по трассе	+	2; 2	266	77	91	165	30	0,75	**0,62**	0,68

Табл. 4. Результаты для детектора целочисленного переполнения.
Table 4. Results for integer overflow checker.

Модель	АСД	Символы	Резюме	W_a; W_b	TP	TN	FP	FN	UC	P	R	F_1
-	До фильтрации результатов				152	0	114	0	0	0,57	1	0,73
Phind	-	-	-	300; 2	96	48	66	56	0	0,59	0,63	0,61
	-	-	+	300; 2	98	51	63	54	0	0,61	0,64	0,63
	+	-	-	2; 2	99	54	58	51	4	0,63	0,66	0,64
	+	по трассе	-	2; 2	80	67	46	70	3	0,63	0,53	0,58
	+	по трассе	+	2; 2	82	78	34	68	4	**0,7**1	0,55	0,62
	+	рекурсивно	+	20; 2	82	72	42	67	3	0,66	0,55	0,6
R1-Qwen	-	-	-	300; 2	122	71	40	29	4	0,75	0,81	0,78
	-	-	+	300; 2	120	72	40	31	3	0,75	0,79	0,77
	+	-	-	2; 2	122	68	44	28	4	0,73	0,81	0,77
	+	по трассе	-	2; 2	120	74	39	30	3	0,75	0,8	0,78
	+	по трассе	+	2; 2	121	77	37	28	3	**0,77**	**0,81**	0,79
	+	рекурсивно	+	20; 2	121	69	45	27	4	0,73	0,82	0,77

Для детектора разыменования значения null лучшие результаты фильтрации были получены рассуждающей моделью с использованием АСД и резюме, а также с добавлением кода вдоль трассы предупреждения. Использование такой конфигурации позволило поднять точность на 11 процентных пунктов до 0,85 при полноте 0,89.

Для детектора утечки ресурсов при аналогичных параметрах удалось повысить точность на 4 процентных пункта до 0,91 при полноте 0,97.

В целом можно заметить, что за исключением детектора целочисленного переполнения, для рассуждающей модели использование резюме и извлечения символов вдоль трассы помогало поднять полноту на 8-11 процентных пунктов по сравнению с подходом, использующим только исходный код.

В табл. 5 приводятся результаты сравнения пяти моделей, перечисленных в начале раздела, на объединённом наборе данных, включающем все три типа предупреждений без детектора недостижимого кода. Параметры генерации запросов для каждого типа предупреждения соответствуют последней строке табл. 1, 2 и 4. Можно отметить, что из опробованных вариантов наилучших результатов удалось достичь при помощи рассуждающих моделей – при этом модели с 32 и 70 миллиардами параметров продемонстрировали схожие показатели F_1-меры при росте точности на 10 и 12 процентных пунктов соответственно.

Табл. 5. Сравнение результатов моделей различных типов и размеров.
Table 5. Comparison of result with models of different kinds and sizes.

Модель	Млрд. параметров	TP	TN	FP	FN	UC	P	R	F_1
До фильтрации результатов		857	0	283	0	0	0,75	1	0,86
DeepSeek-R1-Distill-Qwen-32B	32	763	143	137	87	10	0,85	**0,9**	0,87
DeepSeek-R1-Distill-Llama-70B	70	734	171	109	115	11	**0,87**	0,86	0,87
Phind-CodeLlama-34B-v2	34	711	105	177	143	4	0,8	0,83	0,82
DeepSeek-R1-Distill-Qwen-1.5B	1.5	661	79	199	170	31	0,77	0,8	0,78
Qwen2.5-Coder-32B-Instruct	32	540	201	82	314	3	0,87	0,63	0,73

Этап генерации запросов на основе исходного кода и сохранённых данных является общим для всех моделей и занимает несколько минут (в среднем менее секунды на 1 запрос). Время обработки запросов LLM зависит от размера модели, а также от количества токенов как в самом запросе, так и в ответе модели. Используемая библиотека vLLM позволяет объединять запросы в группы, выполняя их обработку параллельно при наличии достаточных ресурсов видеокарты. На этом наборе данных средний размер запроса составляет порядка тысячи токенов, меньшая из моделей завершает их обработку за 10 минут (2 запроса в

секунду), моделям среднего размера требуется около часа (3 секунды на один запрос), а для модели с 70 миллиардами параметров время обработки запросов увеличивается примерно до 2,5 часов (7 секунд на один запрос), а также требуется дополнительная видеокарта для работы.

Ответы моделей были просмотрены вручную, из них были выделены основные причины, по которым происходит некорректная классификация предупреждений:

1) Ошибочные рассуждения модели. Модель может делать некорректные выводы и игнорировать факты, очевидные из предоставленного контекста. LLM могут делать неверные предположения о свойствах переменных, путать пути выполнения, ошибочно интерпретировать информацию из резюме и галлюцинировать.
2) Нехватка или некорректность предоставленного контекста. Модель может неверно классифицировать предупреждения, если в контексте не содержатся важные для их понимания определения или использования переменных, резюме методов или типов. Реже на ответ модели влияют упрощённые предикаты и некорректная информация из резюме.
3) Различия в определении дефекта. В спорных ситуациях ответы модели и эксперта о том, нужно ли выдавать рассматриваемое предупреждение, могут расходиться из-за различий в критериях истинности предупреждений, которые могут отличаться в разных моделях и разных статических анализаторах. Для преодоления этой проблемы можно помещать в запрос критерии истинности предупреждения в виде набора инструкций.
4) Игнорирование моделью явно прописанных требований к ответу.
5) Ошибки эксперта при ручной разметке предупреждений анализатора.

Также в рамках данной работы было опробовано обучение низкорангового адаптера LoRA (Low-Rank Adaptation) для модели Phind на подмножестве предупреждений типа «утечка ресурсов». Разбиение набора предупреждений на обучающую и тестовую выборку производилось попроектно, чтобы избежать использования в тестировании примеров, схожих с теми, которые использовались при обучении. Обучающая выборка состоит из 278 примеров, а тестовая – из 150. Хотя такой размер выборки считается небольшим, мы посчитали его пригодным для дообучения, поскольку данная LoRA предназначена для проверки одного конкретного типа ошибки с фиксированными шаблонами вопроса и ответа. Для более сложных задач потребовалось бы увеличить размер набора данных, что представляет собой проблему из-за необходимости ручной разметки предупреждений анализатора на реальных проектах.

Запросы для обучающего набора составлялись по шаблону с абстрактными синтаксическими деревьями, добавлением символов по трассе и резюме функций. В качестве эталонного ответа использовалось только слово “Yes” или “No” без промежуточных рассуждений – формулировка запроса была изменена соответствующим образом. Обучение выполнялось с константой альфа, равной 128 и рангом адаптера, равным 64, продолжалось от 4 до 8 эпох, в зависимости от строки таблицы, и заняло порядка 100 минут для 8 эпох обучения. Результаты оценки приведены в табл. 6. Базовая точность анализатора отличается от указанной в табл. 1 из-за того, что оценка происходит только на тестовой выборке. Результаты модели без обучения, с которой происходит сравнение, также соответствуют запросу, требующему односложный ответ.

После дообучения наилучшим результатом для модели Phind стал прирост точности на 3 процентных пункта до 0,81 при полноте 0,94, что превосходит результаты этой же модели без использования LoRA.

Дообучение рассуждающих моделей представляет бóльшую сложность, поскольку для каждого предупреждения в обучающей выборке нужно указать не только сам ответ, но и

цепочку рассуждений, к нему приводящую. Обучающие примеры таких рассуждений можно получать при помощи моделей с бóльшим количеством весов (в том числе проприетарных) и отбирать из них те примеры, в которых модель дала правильный ответ. Чтобы повысить количество пригодных примеров, можно в изначальном запросе подсказывать правильный ответ и просить рассуждать так, будто модель догадалась до него самостоятельно. Более технически сложным вариантом является обучение с подкреплением, для которого не нужны эталонные примеры рассуждений, а оценивается именно итоговый ответ модели.

Табл. 6. Результаты для детектора утечки ресурсов с использованием LoRA.
Table 6 Results for resource leak checker with use of LoRA.

Модель	АСД	Символы	Резюме	W_a; W_b	Кол-во эпох	TP	TN	FP	FN	P	R	F_1
До фильтрации результатов						116	0	34	0	0,77	1	0,87
Phind	+	по трассе	+	2; 2	-	106	4	30	10	0,78	0,91	0,84
Phind + LoRA					4	116	2	32	0	0,78	1	0,88
					6	109	8	26	7	**0,81**	**0,94**	0,87
					8	105	10	24	11	0,81	0,91	0,86

7. Заключение

В статье описан метод повышения точности результатов статического анализа за счёт фильтрации предупреждений большой языковой моделью с использованием вспомогательной информации, извлечённой из анализатора, который был реализован в рамках статического анализатора SharpChecker. Для этого в статическом анализаторе был разработан модуль для сбора и сохранения необходимой информации, а также модуль генерации запросов к LLM и интерпретации ответов модели на основе собранной информации.

Метод был протестирован на размеченных вручную предупреждениях статического анализатора SharpChecker четырёх типов при помощи двух LLM с открытыми весами. Было показано, что предложенный подход позволяет повысить точность детектора ошибок целочисленного переполнения с 57% до 77% (прирост в 20%) при полноте 80%, детектора разыменований null – до 85% (прирост в 11%) при полноте 89%, детектора утечек ресурсов – до 91% (прирост в 4%) при полноте 97%. Использование вспомогательной информации, собранной статическим анализатором, повышает полноту фильтрации предупреждений как минимум на 8%, а для детектора недостижимого кода – до 27%.

Было проведено сравнение пяти моделей разных типов («рассуждающие» и обычные) и размеров (от 1,5 миллиардов параметров до 70) на наборе данных, объединяющем 3 типа предупреждений.

Кроме того, было опробовано дообучение LLM при помощи низкорангового адаптера LoRA, которое позволило повысить точность и полноту фильтрации ошибок типа «утечка ресурсов» на 3% (до 81% и 94%, соответственно).

Список литературы / References

[1]. Gerasimov A. Y. Directed dynamic symbolic execution for static analysis warnings confirmation. Programming and Computer Software, vol. 44, 2018, pp. 316-323. DOI: 10.1134/S036176881805002X

[2]. Tsiazhkorob U. V., Ignatyev V. N. Classification of Static Analyzer Warnings using Machine Learning Methods. Ivannikov Memorial Workshop (IVMEM), IEEE, 2024, pp. 69-74. DOI: 10.1109/IVMEM63006.2024.10659704.

[3]. Ignatyev V. N., Shimchik N. V., Panov D. D., Mitrofanov A. A. Large language models in source code static analysis. Ivannikov Memorial Workshop (IVMEM), IEEE, 2024, pp. 28-35. DOI: 10.1109/IVMEM63006.2024.10659715.

[4]. GPT-4 | OpenAI, available at: https://openai.com/index/gpt-4/, accessed 14.05.2025.

[5]. Koshelev V. K., Ignatiev V. N., Borzilov A. I., Belevantsev A. A. SharpChecker: Static analysis tool for C# programs. Programming and Computer Software, vol. 43, 2017, pp. 268-276. DOI: 10.1134/S0361768817040041.

[6]. Ivannikov V. P., Belevantsev A. A., Borodin A. E., Ignatiev V. N., Zhurikhin D. M., Avetisyan A. I. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, 2014, pp. 265-275. DOI: 10.1134/S0361768814050041.

[7]. Li H., Hao Y., Zhai Y., Qian Z. Enhancing static analysis for practical bug detection: An llm-integrated approach. Proceedings of the ACM on Programming Languages, vol. 8, No. OOPSLA1, 2024, pp. 474-499. DOI: 10.1145/3649828.

[8]. Mohajer M. M., Aleithan R., Harzevili N. S., Wei M., Belle A. B., Pham H. V., Wang S. Effectiveness of ChatGPT for static analysis: How far are we? Proceedings of the 1st ACM International Conference on AI-Powered Software, 2024, pp. 151-160. DOI: 10.1145/3664646.3664777.

[9]. Wen C., Cai Y., Zhang B., Su J., Xu Z., Liu D., Qin S., Ming Z., Cong, T. Automatically inspecting thousands of static bug warnings with large language model: How far are we? ACM Transactions on Knowledge Discovery from Data, vol. 18, No. 7, 2024, pp. 1-34. DOI: 10.1145/3653718.

[10]. Li Z., Dutta S., Naik M. IRIS: llm-assisted static analysis for detecting security vulnerabilities. arXiv preprint arXiv:2405.17238, 2024.

[11]. Khare A., Dutta S., Li Z., Solko-Breslin A., Alur R., Naik M. Understanding the effectiveness of large language models in detecting security vulnerabilities. 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE, 2025, pp. 103-114. DOI: 10.1109/ICST62969.2025.10988968.

[12]. Introduction – Tree-sitter, available at: https://tree-sitter.github.io/tree-sitter/, accessed 14.05.2025.

[13]. Mou L., Li G., Zhang L., Wang T., Jin Z. Convolutional neural networks over tree structures for programming language processing. Proceedings of the AAAI conference on artificial intelligence, vol. 30, No. 1, 2016. DOI: 2016.10.1609/aaai.v30i1.10139.

[14]. GitHub – The Roslyn .NET compiler, available at: https://github.com/dotnet/roslyn, accessed 14.05.2025.

[15]. GitHub – mozilla/dxr, available at: https://github.com/mozilla/dxr, accessed 14.05.2025.

[16]. vllm-project/vllm: A high-throughput and memory-efficient inference and serving engine for LLMs, available at: https://github.com/vllm-project/vllm, accessed 14.05.2025.

Информация об авторах / Information about authors

Данила Дмитриевич ПАНОВ – старший лаборант ИСП РАН, студент ВМК МГУ. Его научные интересы включают статический анализ программного обеспечения и большие языковые модели.

Danila Dmitrievich PANOV – senior laboratory assistant at ISP RAS, student at CMC faculty of Lomonosov Moscow State University. His research interests include static analysis of programs and large language models.

Никита Владимирович ШИМЧИК – кандидат технических наук, младший научный сотрудник ИСП РАН. Его научные интересы включают статический анализ программного обеспечения, большие языковые модели.

Nikita Vladimirovich SHIMCHIK – Cand. Sci. (Tech.), researcher at ISP RAS. His research interests include static analysis of programs, large language models.

Дмитрий Александрович ЧИБИСОВ – аспирант ИСП РАН, Его научные интересы включают статический анализ программного обеспечения, большие языковые модели.

Dmitrii Aleksandrovich CHIBISOV – postgraduate student at ISP RAS. His research interests include static program analysis and large language models.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, член-корреспондент РАН, ведущий научный сотрудник ИСП РАН, профессор кафедры системного программирования ВМК МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., corresponding Member RAS, leading researcher at ISP RAS, Professor at Moscow State University. Research interests: static analysis, program optimization, parallel programming.

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

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

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

Предсказание истинности предупреждений промышленного статического анализатора с использованием методов машинного обучения

[1,3] *У.В. Тяжкороб, ORCID: 0000-0002-2375-6842 <tsiazhkorob@ispras.ru>*
[1,2] *М. В. Беляев, ORCID: 0000-0003-3489-3508 <mbelyaev@ispras.ru>*
[1,2] *А.А. Белеванцев, ORCID:0000-0003-2817-0397 <abel@ispras.ru>*
[1,2] *В. Н. Игнатьев, ORCID: 0000-0003-3192-1390 <valery.ignatyev@ispras.ru>*

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

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

[3] *Московский физико-технический институт, 141701, Россия, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*

Аннотация. В данной работе описан механизм автоматической классификации предупреждений статического анализа с использованием методов машинного обучения. Статический анализ является инструментом поиска потенциальных уязвимостей и ошибок в исходном коде. Однако зачастую статические анализаторы генерируют большое количество предупреждений, причем как истинных, так и ложных. Вручную проанализировать все найденные анализатором дефекты является трудоемкой и времязатратной задачей. Разработанный механизм автоматической классификации показал высокую точность более 93% при полноте около 96% на наборе предупреждений, сгенерированных промышленным инструментом статического анализа Svace при анализе реальных проектов. Генерация набора данных для модели машинного обучения основана на предупреждениях и метриках исходного кода, полученных в процессе анализа проекта статическим анализатором. В работе рассматриваются различные подходы к отбору и обработке признаков классификатора с учетом различных особенностей рассматриваемых алгоритмов машинного обучения. Эффективность работы механизма и его независимость от языка программирования позволили добавить его в промышленный инструмент статического анализа Svace. Были рассмотрены различные подходы к интеграции инструмента, учитывающие специфику статического анализатора, и выбран наилучший из них.

Ключевые слова: машинное обучение; статический анализ; классификация; метрики исходного кода; предупреждения.

Для цитирования: Тяжкороб У.В., Беляев М. В., Белеванцев А.А., Игнатьев В.Н. Предсказание истинности предупреждений промышленного статического анализатора с использованием методов машинного обучения. Труды ИСП РАН, том 37, выпи. 6, часть 1, 2025 г., стр. 101–120. DOI: 10.15514/ISPRAS–2025–37(6)–6.

Machine Learning-Based Validation of Warnings in an Industrial Static Code Analyzer

[1,3] *U.V. Tsiazhkorob, ORCID:0000-0002-2375-6842 <tsiazhkorob@ispras.ru>*
[1,2] *M.V. Belyaev, ORCID: 0000-0003-3489-3508 <mbelyaev@ispras.ru>*
[1,2] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*
[1,2] *V.N. Ignatiev, ORCID: 0000-0003-3192-1390 <valery.ignatyev@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, 1, Leninskie Gory, Moscow, 119991, Russia.*

[3] *Moscow Institute of Physics and Technology, 9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

Abstract. This paper describes a mechanism for the automatic classification of static analysis warnings using machine learning methods. Static analysis is a tool for detecting potential vulnerabilities and bugs in source code. However, static analyzers often generate a large number of warnings, including both true and false positives. Manually analyzing all the defects found by the analyzer is a labor-intensive and time-consuming task. The developed automatic classification mechanism demonstrated high precision of more than 93% with a recall of about 96% on a set of warnings generated by the industrial static analysis tool Svace during the analysis of real-world projects. The dataset for the machine learning model is generated based on the warnings and source code metrics obtained during the static analysis of the project. The paper explores various approaches to feature selection and processing for the classifier, taking into account the characteristics of different machine learning algorithms. The mechanism's efficiency and its independence from the programming language allowed it to be integrated into the industrial static analysis tool Svace. Various approaches to integrating the tool were considered, accounting for the specifics of the static analyzer, and the most convenient one was selected.

Keywords: machine learning; static analysis; classification; source code metrics; warnings.

For citation: Tsiazhkorob U.V., Belyaev M.V., Belevantsev A.A., Ignatiev V.N. Machine learning-based validation of warnings in an industrial static code analyzer. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1 , 2025, pp. 101-120 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-6.

1. Введение

Использование статического анализа исходного кода в промышленной разработке становится стандартной практикой. Это позволяет значительно повысить качество и безопасность программного обеспечения, снижая затраты на поддержку создаваемых программных продуктов. Однако статические анализаторы кода могут генерировать как истинные, так и ложные предупреждения. По этой причине для оценки качества работы статического анализатора принято использовать точность – долю истинных предупреждений среди всех, выданных анализатором. Точность популярных статических анализаторов варьируется от 50% до 100%, в зависимости от типа искомой уязвимости. Современные статические анализаторы способны выявлять сотни различных типов уязвимостей и ошибок в коде, что приводит к генерации большого количества предупреждений для крупных проектов (в среднем порядка 10 предупреждений на 1000 строк исходного кода). Поскольку сгенерированные предупреждения могут быть как истинными, так и ложными, необходимо производить анализ результатов, что чаще всего разработчикам приходится делать вручную. Это значительно усложняет и замедляет процесс разработки программного обеспечения. Таким образом, актуальность разработки механизма автоматической классификации предупреждений на истинные и ложные обусловлена потребностью в повышении эффективности разработки программного обеспечения и отсутствием существующих

масштабируемых, проверенных и интегрированных в промышленные статические анализаторы решений. Возможность автоматической классификации предупреждений статического анализа уже была показана в работе [1]. Однако во время и после интеграции данного механизма в статический анализатор **Svace** [2] были выявлены особенности статического анализатора и анализируемых языков программирования, учитывать которые необходимо для корректной классификации предупреждений.

Целью данной работы является доработка и повышение точности механизма автоматической классификации выданных статическим анализатором предупреждений с использованием методов машинного обучения и его интеграция в промышленный инструмент статического анализа **Svace**.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- найти и проанализировать существующие проблемы в работе механизма;
- доработать схему интеграции со статическим анализатором **Svace**, учитывая особенности работы анализатора и различия в анализе разных языков программирования;
- повысить качество и эффективность классификации с помощью отбора признаков и выбора наилучшего алгоритма машинного обучения;
- рассмотреть различные подходы к интеграции механизма в **Svace** с поддержкой различных сценариев использования и реализовать наиболее оптимальный из них.

Ключевым требованием к работе механизма классификации является повышение точности результатов работы статического анализатора без потерь в полноте.

- поддержка форматов представления данных, полученных в процессе работы статического анализатора;
- эффективная обработка больших объемов данных, включая десятки тысяч предупреждений и сотни различных метрик, собранных из миллионов строк кода;
- обработка результатов работы анализатора с учетом особенностей их структуры и различий для разных языков программирования;
- возможность дообучения модели на пользовательских данных;
- компактное хранение модели, позволяющее включить её в состав дистрибутива статического анализатора.

2. Существующие решения

Для автоматической классификации предупреждений, полученных от статических анализаторов, существует ряд решений и исследований.

Одно из них основано на использовании сверточных нейронных сетей (CNN) [3]. В этом решении классификатор использует фрагменты исходного кода в качестве признаков модели машинного обучения. Средняя точность решения достигает 79.72%. Однако представлены результаты только для шести детекторов, для которых удалось выявить лексические шаблоны исходного кода, ведь для большинства дефектов недостаточно локального контекста для определения истинности. Более того, характерной особенностью моделей на основе сверточных нейронных сетей является длительное время обучения (порядка нескольких часов) даже при использовании графического ускорителя.

Также существуют исследования по применимости машинного обучения для классификации предупреждений статических анализаторов [4]. В данной работе рассматривается применение алгоритмов машинного обучения, таких как метод опорных векторов (SVM) [5], K-ближайших соседей (KNN) [6], случайный лес (Random Forest) [7, 8] и алгоритм RIPPER

[9], для классификации предупреждений статических анализаторов. Наилучшую точность показал алгоритм случайного леса (83%–98%).

В качестве признаков для моделей машинного обучения в данной работе используются 111 метрик исходного кода, рассчитываемые с помощью механизма Understand [10]. Тестирование производилось только для 7 типов ошибок из перечня общих дефектов и уязвимостей CWE [11]. В качестве набора предупреждений использовались результаты работы статического анализатора на синтетически сгенерированной выборке Juliet для C++ [12].

Также существует решение с инкрементальным механизмом машинного обучения. Тестирование производилось на 9 проектах с открытым исходным кодом на языке программирования Java. В качестве признаков использовались характеристики исходного кода и предупреждений, полученные с помощью статического анализа кода, метаданных проекта и истории версий. Были исследованы три модели машинного обучения, при инкрементальном активном обучении наилучшую точность показал метод опорных векторов (около 90%). Однако данное решение плохо масштабируемо и ресурсозатратно, поскольку для каждого нового проекта необходимо снова запускать активное обучение, и требуется контроль качества на каждом этапе обучения модели.

Существующие исследования отражают лишь теоретическую возможность использования методов машинного обучения для верификации результатов работы статических анализаторов, а существующие решения имеют ряд недостатков и ограничений. Однако с помощью анализа существующих решений можно сделать выводы о важности определенных признаков и качестве определенных моделей машинного обучения, что позволяет проводить исследования, опираясь на наилучшие практики.

В данной работе представлен подход к автоматической классификации предупреждений, учитывающий все недостатки существующих решений. Механизм не зависит от синтаксиса языка программирования и типа дефекта, классификация производится за малое по сравнению со статическим анализом время. При тестировании на реальных проектах с открытым исходным кодом была достигнута высокая точность, составляющая 92%.

3. Разработанный механизм

Разработанный алгоритм автоматической классификации предупреждений состоит из 4 этапов:

- генерация признаков: создание и обработка признаков для модели машинного обучения на основе предупреждений и метрик исходного кода;
- обучение модели: подбор гиперпараметров модели и обучение на предоставленном наборе данных;
- классификация предупреждений статического анализатора: модель предсказывает вероятность истинности предупреждений;
- отображение или фильтрация: результаты классификации отображаются в пользовательском интерфейсе или используются для фильтрации ложных предупреждений.

В качестве признаков для модели машинного обучения используются метрики исходного кода. *Метриками* [13] исходного кода называются числовые значения, характеризующие качество исходного кода и широко применяемые для обеспечения качества программного обеспечения (QA) [14], например, цикломатическая сложность, то есть количество линейно независимых путей в коде. Метрики вычисляются отдельно для каждой директории, файла, класса и метода, которые в работе будем называть *сущностями*. Вычисление метрик исходного кода в **Svace** осуществляется компонентом **SCRA** (**Source Code Relation Analyzer**) [15]. В результате работы **Svace** генерируются файл с информацией о метриках и

файл с информацией о сущностях. Для каждой сущности анализируемого проекта статический анализатор сохраняет имя, числовой уникальный идентификатор и уникальный идентификатор родительской сущности. Для каждой метрики исходного кода сохраняется тип, значение и уникальный идентификатор сущности, к которой относится метрика. Всего **Svace** рассчитывает значения для 141 типа метрик.

Рассмотрим особенности некоторых стадий работы механизма автоматической классификации предупреждений.

3.1 Особенности реализации механизма генерации признаков

Реализация алгоритма сопоставления предупреждения с его уникальным набором значений метрик должна учитывать особенности хранения результатов работы **Svace**. Основная сложность состоит в том, чтобы определить, какие именно значения метрик соответствуют конкретному предупреждению. Поэтому, сперва необходимо определить, какие сущности относятся к предупреждению по полному пути к предупреждению и сигнатуре функции. После этого становится возможным по уникальному идентификатору сущности найти все соответствующие ей значения метрик, тем самым сопоставив предупреждения со специфичным для него набором метрик исходного кода.

3.1.1 Обобщенный алгоритм

Схема алгоритма генерации признаков представлена на рис. 1.

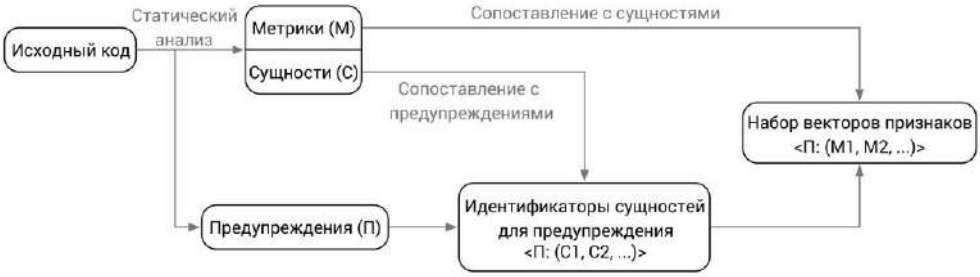


Рис. 1. Схема алгоритма генерации признаков.
Fig. 1. Feature generation scheme.

После составления списка сущностей, относящихся к предупреждению, происходит поиск значений метрик для каждой сущности из списка. Как известно, в полном пути к файлу может быть несколько директорий, аналогично, в сигнатуре функции может быть несколько классов. В вектор признаков записываются метрики, вычисленные для наиболее глубоких сущностей каждого типа. То есть из всех директорий метрики записываются для той, в которой непосредственно находится файл. Помимо метрик, в файл с признаками записывается также тип найденного дефекта, статус предупреждения, язык программирования анализируемого проекта и порядковый номер предупреждения (для обратного сопоставления признаков с исходными предупреждениями).

Для ускорения составления списков идентификаторов сущностей используется кэширование: при полном или частичном совпадении искомого пути или сигнатуры функции с предыдущим предупреждением нет необходимости искать все сущности заново. Для использования кэширования после прочтения файла с предупреждениями информация о них сортируется по имени файла. Если имена файлов одинаковые, то сортировка происходит по имени функции.

3.1.2 Особенности реализации

В процессе разработки и работы механизма генерации признаков были выявлены особенности хранения результатов работы статического анализатора.

Для того, чтобы получить набор имен сущностей из полного пути к файлу и сигнатуры функции, нужно разделить их строковые представления по соответствующим символам-разделителям. Кроме того, для C++ имена функций являются декорированными (*mangled*), и чтобы получить исходные имена функций, их необходимо восстановить при помощи декодера.

Еще одной проблемой является невозможность однозначно определить сущность только по ее имени, поскольку несколько сущностей могут иметь одинаковые значения имен. Информация о сущностях имеет иерархическую структуру: проекты с точки зрения папок и файлов, исходный код – с точки зрения классов и функций. В качестве решения данной проблемы предлагается искать имена сущностей последовательно с учетом предыдущей сущности как родителя текущей. Чтобы поиск был наиболее эффективным, строится дерево сущностей.

Однако и это решение не гарантирует корректную работу поиска сущностей, поскольку одинаковые имена могут иметь и сущности с одним и тем же родителем, например, в случае с разделяемыми (`partial`) классами на языке C# или для языков C++ и C# при многократной компиляции файла с различными значениями символов условной компиляции. Для решения данной проблемы предложено добавить проход по дереву сущностей после его построения, в процессе которого составляется общий список потомков для сущностей с одинаковыми идентификаторами и именами. Чтобы сохранить исходную иерархию, указатели на родительский узел у всех потомков остаются прежними.

Из-за различия в форматах записи сигнатур функций с вложенными классами для предупреждения (`Outer.Inner.Func()`) и соответствующих им сущностей в файле с информацией о всех сущностях (`Outer` → `Outer$Inner` → `Func`) после построения дерева сущностей необходимо совершать еще один проход для устранения различий.

Описанный инструмент был реализован на языке C++ для уменьшения затрат ресурсов и времени.

3.2 Обработка данных и выбор модели

Механизм классификации реализован в виде программы на языке Python, использующей популярные библиотеки машинного обучения и анализа данных **pandas** [16] и **scikit-learn** [17], а также библиотеку **Beautiful Soup** [18] для работы с данными в формате XML.

3.2.1 Добавление и отбор признаков

В сгенерированном наборе векторов признаков значения типа дефекта и статуса предупреждения являются строковыми. Статус предупреждения кодируется следующим образом: истинное предупреждение кодируется значением 1, ложное предупреждение – значением 0. Тип дефекта определяется уникальным строковым значением. Однако к этим значениям могут быть добавлены строковые значения тегов в любом количестве и в любом порядке. Например, тег `.TEST` в типе предупреждения `HANDLE_LEAK.EXCEPTION.TEST` показывает, что дефект типа `HANDLE_LEAK` найден в коде тестов; в то время как тег `.RET` в типе предупреждения `DEREF_OF_NULL.RET.LIB.PROC` означает, что найденный дефект типа `DEREF_OF_NULL` относится к возвращаемому из функции значению. Список тегов является общим для всех типов дефектов. Список тегов является общим для всех типов дефектов. Поскольку каждое предупреждение имеет ровно один тип дефекта, но может иметь произвольное количество тегов, тип предупреждения кодируется с помощью инструмента `LabelEncoder`, а для тегов применяется тип кодирования One-Hot с помощью инструмента

`OneHotEncoder` библиотеки **scikit-learn**. То есть тип дефекта кодируется уникальным числовым значением, в то время как для каждого тега добавляется свой признак со значением 1, если такой тег добавлен к типу дефекта, и 0, если нет.

Для того, чтобы алгоритм машинного обучения лучше понимал структуру данных и закономерности в них, в качестве признаков добавляются исходная точность анализатора по истинным предупреждениям и общее количество предупреждений в файле и в функции. Исходная точность рассчитывается на наборе данных для обучения как доля истинных предупреждений от всего количества предупреждений, а количество предупреждений в файлах и в функциях рассчитывается на полном наборе данных. Создание новых признаков, специфичных для данной задачи, значительно повышает информативность данных. Общее количество признаков, включающих в себя метрики исходного кода, тип дефекта, количество предупреждений в файле и в функции и исходную точность детектора, равно 163.

При анализе накопленной дисперсии осей проекции (рис. 2) было выявлено, что при количестве признаков больше 100 этот показатель перестает расти. Поскольку накопленная дисперсия осей проекции показывает, насколько хорошо описывается изначальное распределение величин в зависимости от количества признаков, то при уменьшении размерности исходных признаков до 100 исходные зависимости не будут потеряны.

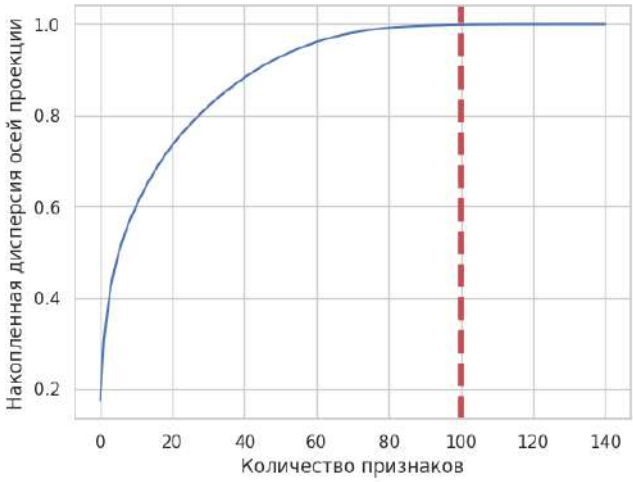


Рис. 2. Накопленная дисперсия осей проекции.
Fig. 2. Cumulative explained variance.

Отбор признаков производить важно, поскольку большое количество признаков может привести к переобучению: чем больше признаков, тем проще модели найти ложные зависимости в данных. Более того, при большей размерности пространства признаков увеличивается время обучения и время классификации, особенно для алгоритмов машинного обучения, использующих деревья, а также алгоритмов, основанных на градиентном бустинге.

3.2.2 Подготовка наборов данных

После создания набора векторов признаков для предупреждений, для обучения модели необходимо разделить набор данных на тренировочный, валидационный, и, в случае тестирования, тестовый. Обычно тренировочный набор данных составляет 80% от всех векторов, а валидационный, соответственно, 20%. В случае, когда набор данных делится на три части, тренировочный набор данных составляет 56%, валидационный – 14% и тестовый – 30%. Чтобы в тренировочный набор данных попали предупреждения для всех типов дефектов, разделение исходного набора данных производится отдельно для каждого типа

дефекта. Затем полученные данные объединяются в общий тренировочный, валидационный и, в случае тестирования, тестовый набор. Исходный набор данных является несбалансированным по количеству ложных и истинных предупреждений: количество объектов класса 0 в несколько раз меньше количества объектов класса 1. Чтобы сохранить отношения количества объектов класса 0 к количеству объектов класса 1 как во всех наборах данных, разделение исходного набора данных производится со стратификацией, то есть с балансировкой. Функция `train_test_split` библиотеки **scikit-learn**, которая является механизмом разбиения исходного набора данных, имеет параметр `stratify`, значением которого является признак для стратификации, в данном случае это целевое значение – статус предупреждения.

Важным этапом подготовки данных для модели машинного обучения является масштабирование – приведение распределения значений векторов признаков к среднему, равному 0, и стандартному отклонению, равному 1. Необходимость масштабирования обусловлена тем, что данные, разные по физическому смыслу, сильно отличаются по абсолютным значениям. Дисбаланс между значениями признаков негативно влияет на построение модели, приводя к смещенным результатам предсказаний, ошибкам классификации и, соответственно, снижению точности.

3.2.3 Анализ важности признаков для моделей машинного обучения

В качестве моделей машинного обучения рассматриваются классификаторы **CatBoost** [19], **Random Forest** [20] и **XGBoost** [21]. **CatBoost** и **XGBoost** используют алгоритмы градиентного бустинга. **CatBoost** является алгоритмом машинного обучения с высокой производительностью и использует разнообразные стратегии регуляризации для предотвращения переобучения.

Для уже обученных моделей с настроенными гиперпараметрами (настройка подробно описывается в статье [1]) по отдельности был произведен анализ важности признаков с помощью атрибута `feature_importances_`, рассчитываемого автоматически методами использованных библиотек после обучения модели. Первые 15 важных признаков для каждой модели представлены на графиках (рис. 3).

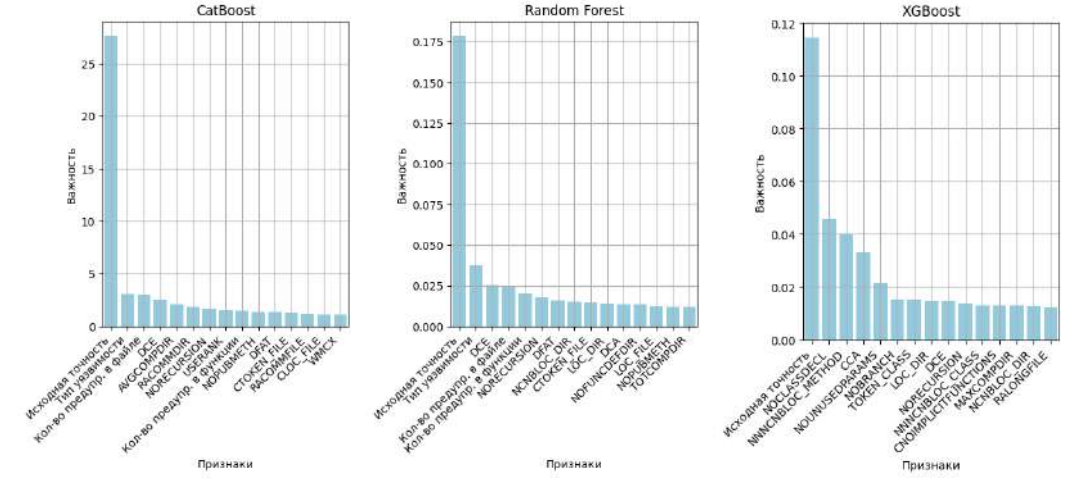


*Рис. 3. Важность признаков для **CatBoost**, **Random Forest** и **XGBoost** соответственно.*
*Fig. 3. Feature importances for **CatBoost**, **Random Forest** and **XGBoost** respectively.*

Из анализа графиков можно сделать вывод, что наиболее важным признаком для всех трех моделей является изначальная точность по истинным предупреждениям. Как уже было

упомянуто ранее, именно этот признак показывает, насколько хорошо работает детектор. Чем выше этот показатель, тем выше вероятность того, что предупреждение является истинным. Также важными признаками для моделей оказались количество предупреждений в файле и количество предупреждений в функции. Еще одним важным признаком для каждой модели является внешняя связность (DCE) [22]. Любое несогласованное изменение во внешних зависимостях может привести к ошибкам исполнения кода текущего модуля. Чем больше внешних зависимостей, тем больше вероятность таких изменений и, соответственно, истинности предупреждения. Также важными признаками являются NORECURSION (количество рекурсивных функций в файле), NOPUBMETH (количество публичных методов в классе), DFAT (количество зависимостей между классами каталога), CTOKEN_FILE (количество токенов в комментариях в файле), NCNBLOC_DIR (количество непустых строк кода в директории, не включая комментарии), LOC_DIR (количество строк кода в директории).

4. Интеграция механизма классификации предупреждений в статический анализатор Svace

Чтобы предоставить пользователям **Svace** доступ к функциональности классификации предупреждений, необходимо интегрировать механизм классификации в существующую инфраструктуру статического анализатора [23]. Статический анализатор **Svace** состоит из нескольких компонентов: подсистемы перехвата сборки, подсистемы анализа, включающей в себя большое количество независимых модулей (собственный многоязыковой движок символьного выполнения и анализа потоков данных, **Clang Static Analyzer** [24], **SpotBugs** [25], **SharpChecker** [26] и т.д.), а также **сервера истории** статического анализа. Сервер истории осуществляет хранение результатов статического анализа в базе данных и предоставляет возможности просмотра предупреждений и исходного кода в веб-интерфейсе, задания статусов предупреждений (не размеченное, истинное, ложное) и комментариев, а также автоматического сопоставления предупреждений между различными запусками статического анализатора, чтобы одни и те же предупреждения не приходилось просматривать повторно. При интеграции механизма классификации необходимо обеспечить выполнение всех этапов работы механизма (вычисление метрик, генерацию признаков, дообучение модели, выполнение классификации, загрузку результатов классификации на сервер для просмотра) наиболее удобным для использования способом, требующим наименьшего количества дополнительных действий со стороны пользователя.

4.1 Сценарии использования

Существует два основных сценария использования статического анализатора: проверка проекта «с нуля» и регулярное использование в процессе разработки программного продукта, при котором в базе данных сервера истории находится большое количество размеченных предупреждений проекта. Данные сценарии использования **Svace** задают два аналогичных сценария использования механизма классификации предупреждений, которые требуется поддержать – «холодный старт», при котором классификация выполняется при помощи предварительно обученной модели, что позволяет сразу начать использование механизма классификации, и классификацию с дообучением модели на пользовательской разметке, загружаемой с сервера истории, что позволяет существенно увеличить точность классификации.

4.2 Интеграция в процесс анализа

Вначале была рассмотрена задача «холодного старта». В этом режиме классификация выполняется при помощи предварительно обученной модели, распространяемой в составе дистрибутива **Svace**. Было решено встроить классификацию предупреждений в процесс

анализа после получения файла с предупреждениями от всех компонентов. Модификация процесса анализа показана в табл. 1.

4.3 Взаимодействие с сервером истории для обучения на пользовательских данных

При переходе от классификации при помощи предварительно обученной модели к дообучению на пользовательских данных возникла необходимость взаимодействия с сервером истории, чтобы автоматизировать процесс получения имеющихся на нём размеченных данных. **Svace** поддерживает работу с двумя серверами истории: встроенным сервером, реализованным непосредственно в составе **Svace**, и внешним сервером истории **Svacer** [27], разрабатываемым отдельной командой в ИСП РАН.

Табл. 1. Изменение последовательности действий в процессе анализа для интеграции механизма классификации предупреждений.
Table 1. Changes in the analysis actions order for integration of warnings classification mechanism.

Было	Стало
	вычисление полных метрик для кода на C/C++, Java и базовых метрик для кода на Kotlin, Go, Python
анализ кода на языках C/C++, Java, Kotlin, Go, Python	
анализ кода на C# и Visual Basic .NET и вычисление метрик для кода на C#[1] инструментом **SharpChecker**	
сохранение предупреждений, выданных всеми движками анализа, в единый файл	
вычисление базовых метрик для кода на C/C++, Java, Kotlin, Go, Python	генерация признаков для выданных предупреждений с использованием вычисленных метрик
	классификация полученных предупреждений при помощи предварительно обученной модели
	сохранение предупреждений и результатов классификации в итоговый файл

4.3.1 Встроенный сервер истории

Основной шаблон использования **Svace** со встроенным сервером истории состоит из следующих этапов:

- контролируемая сборка анализируемого проекта для генерации внутреннего представления;
- анализ полученного объекта сборки;
- импорт результатов анализа на встроенный сервер истории;
- просмотр и разметка результатов в веб-интерфейсе сервера истории.

Чтобы автоматизировать все шаги, необходимые для классификации предупреждений с использованием пользовательской разметки, без необходимости дополнительных действий

[1] За время разработки и интеграции механизма классификации в **SharpChecker** было реализовано вычисление метрик также для кода на языке Visual Basic .NET.

от пользователя, классификация была перенесена с этапа анализа на этап импорта на сервер истории, так как только на этом этапе **Svace** получает от пользователя информацию для подключения к серверу. Кроме того, анализ зачастую выполняется на другом компьютере или в изолированном окружении, не имеющем сетевого доступа к серверу истории, что обосновывает данное решение. Схема работы механизма классификации на этапе импорта результатов показана на рис. 4.

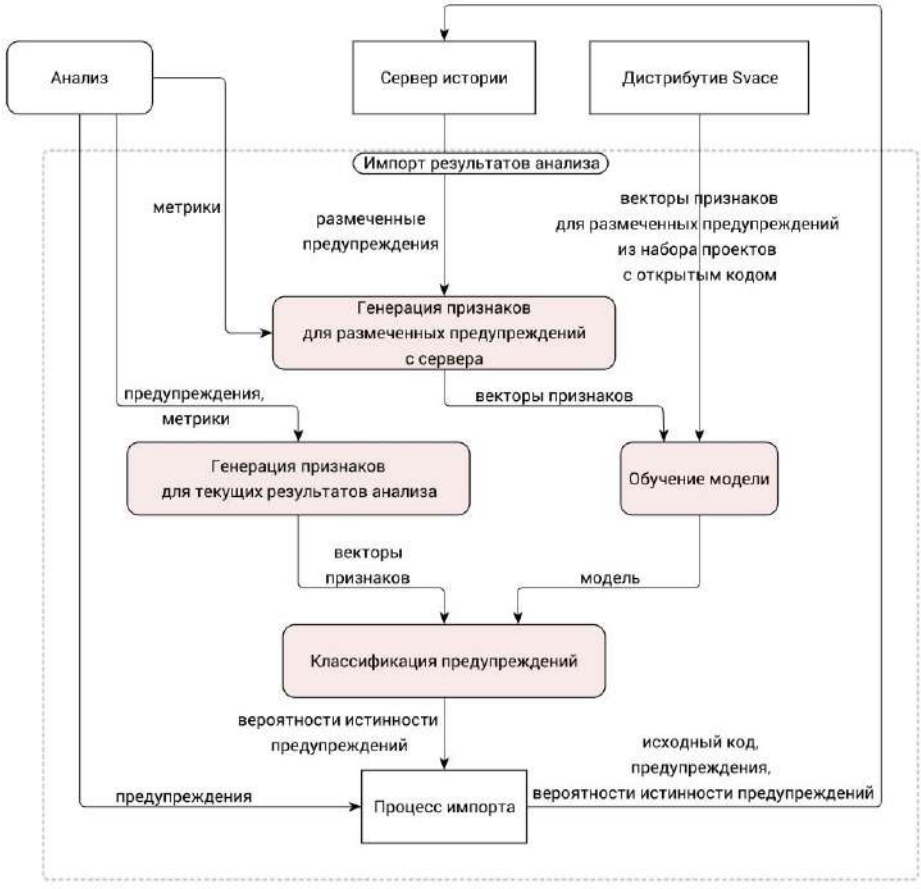


Рис. 4. Схема интеграции механизма классификации предупреждений в этап импорта.
Fig. 4. Scheme of integration of the warning's classification mechanism into the import stage.

Как показано на данной схеме, классификация предупреждений выполняется моделью, обученной на размеченных предупреждениях из анализируемого проекта, а также на предупреждениях из проектов с открытым кодом, размеченных разработчиками **Svace** и экспертами Центра верификации ОС Linux [28]. Предварительно сгенерированные векторы признаков для набора проектов с открытым кодом распространяются в составе дистрибутива **Svace**. Векторы признаков для предупреждений с сервера истории генерируются с использованием метрик, вычисленных в текущем запуске анализатора, так как метрики имеют большой объём, и сервер истории не обладает функциональностью хранения метрик.

Итоговая реализация интеграции механизма классификации предупреждений практически не требует от пользователя дополнительных действий, таких, как запуск дополнительных команд вручную или существенная модификация скриптов запуска статического анализатора

в системе непрерывной интеграции. Достаточно лишь включить вычисление полных метрик на этапах сборки и анализа, и тогда получение размеченных данных, обучение модели и классификация предупреждений будут выполнены автоматически, после чего результаты будут доступны для просмотра на встроенном сервере истории.

4.3.2 Сервер истории Svacer

Сервер истории **Svacer** является отдельным инструментом, разрабатываемым отдельно от **Svace** и не входящим в его состав, что затрудняет процесс внесения изменений и добавления новой функциональности. Это делает нежелательной интеграцию механизма классификации предупреждений непосредственно в процесс импорта результатов анализа в **Svacer**. К тому же, частичное дублирование реализации в двух независимых программных компонентах, написанных на разных языках программирования (Java и Go), существенно увеличило бы трудозатраты на разработку и поддержку и привело бы к неизбежному накоплению различий и неисправленных ошибок. Поэтому было решено не интегрировать запуск классификации предупреждений непосредственно в **Svacer**, а реализовать взаимодействие с сервером **Svacer** через публичный API [29] в виде вспомогательной команды **Svace**, повторно используя реализованный в составе команды импорта запуск процессов генерации признаков, обучения модели и классификации предупреждений.

4.4 Отображение результатов классификации предупреждений в пользовательском интерфейсе

Отображение результатов работы механизма автоматической классификации было внедрено в пользовательский интерфейс для просмотра и анализа предупреждений встроенного сервера истории. Результатами работы механизма являются предсказанные моделью вероятности истинности для каждого предупреждения.

На рис. 5 показано, как выглядит обновленный элемент пользовательского интерфейса.

Каждая строка представляет собой информацию о предупреждении. Проанализированные вручную предупреждения выделены цветом фона: истинные – зелёным, ложные –красным. Вероятность истинности отображается в процентах в правом верхнем углу каждого предупреждения.

Ручной анализ данных предупреждений производился уже после работы механизма автоматической классификации и подтвердил корректность работы механизма для данных предупреждений.

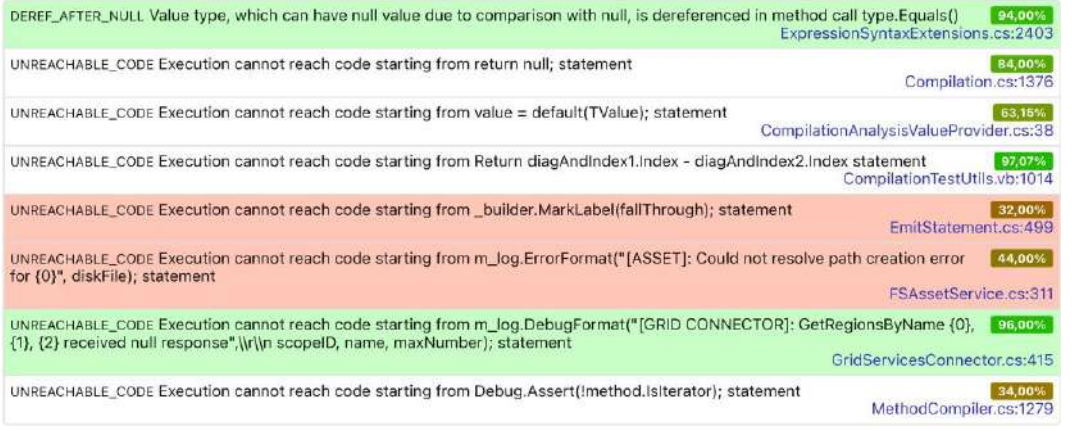


Рис. 5. Фрагмент пользовательского интерфейса встроенного сервера истории.
Fig. 5. Fragment of built-in history server UI.

Пользовательский интерфейс также предоставляет возможность сортировки предупреждений по вероятности истинности, а также фильтрации предупреждений по пороговому значению вероятности, задаваемому пользователем.

В пользовательском интерфейсе **Svacer** в настоящее время поддерживается только отображение вероятности истинности для текущего открытого предупреждения.

5. Результаты тестирования

Для тестирования моделей машинного обучения использовался набор данных, сгенерированных на основе предупреждений, полученных с помощью статического анализа 21 проекта с открытым исходным кодом на языке C#, имеющих суммарный размер более 6 млн. строк кода. Тестирование проводилось на компьютере с процессором AMD Ryzen 5 4600G и 64 ГБ оперативной памяти, работающем под управлением операционной системы Windows 10. Всего было сгенерировано 68182 предупреждения, 13182 из которых были проанализированы и размечены вручную более чем за 5 лет в процессе разработки инструмента статического анализатора. Генерация признаков для всех предупреждений заняла 2 минуты и 20 секунд, в то время как статический анализ этих проектов суммарно занял около 3 часов 30 минут. При процентном разделении набора данных на части, описанном ранее, тренировочный набор данных составляет 8167 векторов признаков, валидационный – 1605, тестовый – 3465.

Время обучения и настройка гиперпараметров для всех моделей классификаторов различается. Для данного набора данных классификатор **CatBoost** обучается около 43 секунд, настройка гиперпараметров и обучение для классификатора **Random Forest** длится 3 минуты и 13 секунд. Для классификатора **XGBoost** подготовка модели происходит дольше – около 20 минут. Так происходит из-за особенностей классификатора, для нормальной работы классификатора **XGBoost** требуется более тонкая настройка, чем для двух других моделей машинного обучения.

В табл. 2 представлены метрики качества бинарной классификации для результатов работы моделей на описанном наборе данных без отбора признаков.

Из таблицы видно, что объектов для истинных предупреждений все метрики выше, чем для ложных предупреждений. Это как раз связано с дисбалансом классов в существующем наборе данных. Чтобы равноценно классифицировать как истинные предупреждения, так и ложные, необходимо помимо точности рассматривать другие метрики качества для выбора подхода и модели машинного обучения. Однако стоит заметить, что показатель `AUC-ROC` (площадь под кривой ошибок) довольно высокий для всех моделей, что показывает, что каждая модель хорошо отличает класс истинных предупреждений от класса ложных.

Ранее было показано, что при ограничении количества признаков до 100 потери изначальных зависимостей происходить не будет, но при этом сами модели станут проще, время обучения станет меньше и, возможно, точность работы моделей может увеличиться. Метрики бинарной классификации для результатов работы моделей машинного обучения на наборах данных с первыми 100 по важности признаками для каждой модели машинного обучения по отдельности представлены в табл. 3.

Основное внимание при оценке результатов классификации необходимо уделить F_1-мерам для истинных и ложных предупреждений, поскольку F_1-мера является комбинацией точности и полноты классификации. Для классификатора **CatBoost**, как и для классификатора **Random Forest**, данный показатель вырос как для первого класса, так и для нулевого. Однако для модели **XGBoost** F_1-мера уменьшилась и для первого, и для нулевого класса. Это могло произойти из-за того, что данный классификатор обычно выигрывает за счет того, что каждый уровень дерева из ансамбля использует разные признаки для разбиения данных, то есть при удалении признаков деревья могли стать менее разнообразными, и точность могла упасть.

Табл. 2. Метрики для набора данных с полным списком признаков.
Table 2. Quality metrics for dataset with the full list of features.

Классификатор		CatBoost	Random Forest	XGBoost
Истинные предупреждения	Точность	93,21%	92,15%	93,21%
	Полнота	96,29%	97,36%	96,77%
	F_1	94,73%	94,68%	94,96%
Ложные предупреждения	Точность	84,46%	87,71%	86,16%
	Полнота	74,19%	69,46%	74,05%
	F_1	78,99%	77,53%	79,65%
Сбалансированная полнота		85,24%	83,41%	85,41%
AUC-ROC		96,61%	96,20%	96,77%

Табл. 3. Метрики для набора данных с отобранными признаками.
Table 3. Quality metrics for dataset with selected features.

Классификатор		CatBoost	Random Forest	XGBoost
Истинные предупреждения	Точность	93,73%	93,21%	93,13%
	Полнота	95,96%	96,77%	96,48%
	F_1	94,83%	94,96%	94,77%
Ложные предупреждения	Точность	83,70%	86,16%	85,05%
	Полнота	76,35%	74,05%	73,78%
	F_1	79,86%	79,65%	79,02%
Сбалансированная полнота		86,16%	85,41%	85,13%
AUC-ROC		96,49%	96,37%	96,80%

Все последующие графики будут для моделей **CatBoost** и **Random Forest**, обученных на наборе данных с сокращенным списком признаков, а для модели **XGBoost** – с полным списком признаков.

Визуализировать различимость классов каждой модели можно с помощью графика плотностей распределения классов (рис. 6). По оси абсцисс у графиков распределения плотности классов отложены значения предсказаний модели. То есть с их помощью можно увидеть, где, относительно предсказаний модели, сконцентрированы истинные и ложные предупреждения. Для всех трех моделей пики плотностей истинных и ложных предупреждений различимы и находятся на большом расстоянии. Это и говорит о том, что все три модели упорядочивают большинство пар (элемент класса ложных предупреждений, элемент класса истинных предупреждений) верно.

При оценке точности результатов работы моделей пороговое значение классификации по умолчанию равно 0.5. Однако на графиках плотностей распределения классов видно, что переход от значений, для которых плотность распределения ложных предупреждений выше, чем плотность распределения истинных, к значениям, где плотность распределения

истинных предупреждений выше, чем плотность распределения ложных, происходит при вероятности истинности, большей 0.5.

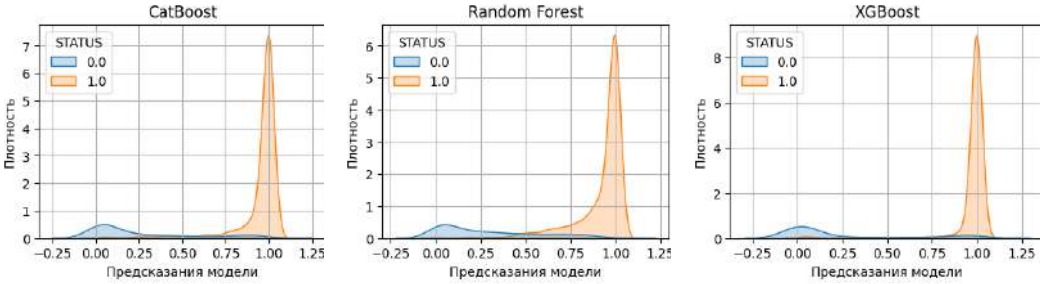


*Рис. 6. Плотность распределения классов для **CatBoost**, **Random Forest** и **XGBoost** соответственно.*
*Fig. 6. Class probability density for **CatBoost**, **Random Forest** and **XGBoost** respectively.*

Чтобы повысить точность работы моделей, можно попробовать изменить пороговое значение классификации. Из графиков также понятно, что при пороговом значении, равном 0.5, полнота по истинным предупреждениям будет выше, чем при пороговых значениях, больших 0.5. Таким образом при выставлении нового значения порогового значения нужно увеличить метрики качества для определения класса 0 и при этом сохранить метрики качества для определения класса 1. Чтобы более сбалансировано выбирать новое пороговое значение, построим графики F_1-мер (рис. 7) для истинных и ложных предупреждений с разными пороговыми значениями классификации.

В качестве пороговых значений берутся числа от 0 до 1 не включительно с шагом 0.01. Верхняя кривая на каждом графике показывает зависимость F_1-меры от порогового значения для истинных предупреждений, нижняя кривая – для ложных предупреждений. Красными горизонтальными прямыми отмечены максимумы данных кривых. Вертикальная прямая отображает пороговое значение. Пороговое значение выбирается так, чтобы F_1-мера была близка к максимуму как для истинных предупреждений, так и для ложных, поскольку с помощью F_1-меры можно найти баланс между точностью и полнотой.

Пороговыми значениями для моделей были выбраны следующие: **CatBoost** – 0.57, **Random Forest** – 0.54, **XGBoost** – 0.79. Метрики качества бинарной классификации с заданными пороговыми значениями представлены в табл. 4. Для всех моделей показатели качества классификации достаточно высокие, однако учитывая качество и время обучения модели, наилучшим классификатором является модель **CatBoost**. При высокой точности в 91.92% удалось добиться высоких показателей полноты, как для класса истинных предупреждений (94.89%), так и для класса ложных предупреждений (80.74%).

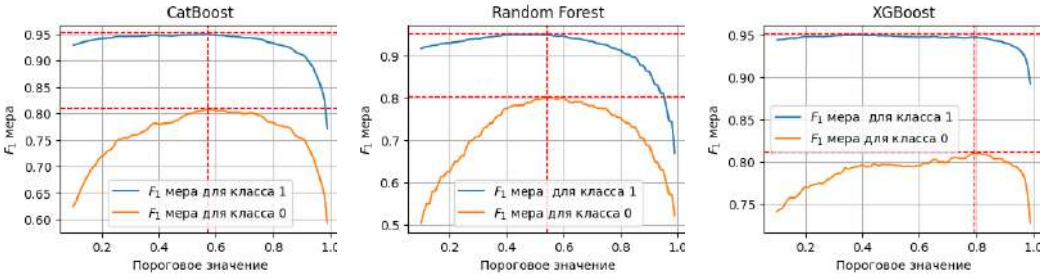


*Рис. 7. F_1-мера для **CatBoost**, **Random Forest** и **XGBoost** соответственно.*
*Fig. 7. F_1-measure for **CatBoost**, **Random Forest** and **XGBoost** respectively.*

Табл. 4. Метрики для моделей с выставленным пороговым значением.
Table 4. Quality metrics for models with threshold.

Классификатор		CatBoost	Random Forest	XGBoost
Истинные предупреждения	Точность	94,44%	93,89%	95,02%
	Полнота	95,34%	95,85%	94,57%
	F_1	94,89%	94,86%	94,79%
Ложные предупреждения	Точность	82,21%	83,46%	80,35%
	Полнота	79,32%	77,03%	81,76%
	F_1	80,74%	80,11%	81,04%
Сбалансированная полнота		87,33%	86,44%	88,16%
AUC-ROC		96,49%	96,37%	96,85%

Матрица ошибок для выбранной модели представлена в табл. 5. В матрице ошибок для классификатора **CatBoost** продемонстрированы абсолютные значения для количества предупреждений и их процентное соотношение для тестового набора данных, размером 3465 предупреждений.

Поскольку разработанный механизм автоматической классификации предупреждений не зависим от языка программирования, его можно применять ко всем языкам программирования, для которых статический анализатор рассчитывает метрики исходного кода, то есть можно получить результаты работы для языков программирования C, C++ и Java.

*Табл. 5. Матрица ошибок для модели **CatBoost**.*
Table 5. Confusion matrix for CatBoost model.

		Действительные	
		Ложные	Истинные
Предсказанные	Ложные	587 (82%)	127 (18%)
	Истинные	153 (6%)	2598 (94%)

Предсказание результатов производилось с помощью классификатора **CatBoost**. Для каждого языка обучалась отдельная модель. Результаты представлены в таблицах 6 и 7.

Для языков C, C++ тестирование производилось на наборе из 4483 предупреждений. В качестве признаков были отобраны первые 100 по важности для модели машинного обучения признаков. С пороговым значением 0.55 была достигнута точность классификации в 82.42%. Точность работы статического анализатора на валидационной выборке равна 68.34%. То есть при фильтрации ложных предупреждений с помощью механизма автоматической классификации точность вырастет на 16.43%. При этом полнота классификации истинных предупреждений – 89.87%.

Для языка Java тестирование производилось на наборе из 1155 предупреждений. В качестве признаков были отобраны первые 80 по важности для модели машинного обучения признаков. С пороговым значением 0.48 была достигнута точность классификации в 85.76%. Точность работы статического анализатора на валидационной выборке равна 61.63%. То есть при фильтрации ложных предупреждений с помощью механизма автоматической

классификации точность вырастет на 25.58%. При этом полнота классификации истинных предупреждений – 90.09%.

Из-за малого количества предупреждений в обучающем наборе данных, достигнуть высоких показателей для классификации ложных предупреждений для данных языков программирования не удалось: F_1-мера для ложных предупреждений составляет 69.67% для C, C++ и 80.93% для Java.

Табл. 6. Матрица ошибок для C/C++ и Java.
Table 6. Confusion matrix for C/C++ and Java.

		Действительные C/C++		Действительные Java	
		Ложные	Ложные	Истинные	Ложные
Предсказанные	Ложные	271 (75%)	91 (25%)	104 (83%)	21 (17%)
	Истинные	145 (15%)	807 (85%)	28 (13%)	191 (87%)

Табл. 7. Метрики для C/C++ и Java.
Table 7. Quality metrics for C/C++ and Java.

Язык программирования		C/C++	Java
Истинные предупреждения	Точность	84,77%	87,21%
	Полнота	89,87%	90,09%
	F1	87,24%	88,63%
Ложные предупреждения	Точность	73,86%	83,20%
	Полнота	65,14%	78,79%
	F1	69,67%	80,93%
Сбалансированная полнота		77,51%	84,44%
AUC-ROC		87,88%	92,38%

6. Заключение

В рамках данной работы были выполнены следующие задачи:

- доработан механизм генерации признаков с учетом выявленных особенностей статического анализатора **Svace** и языков программирования C#, C/C++, Java;
- выбрана модель машинного обучения **CatBoost**, имеющая оптимальный баланс между производительностью и показателями качества классификации для данной задачи;
- метод классификации предупреждений реализован и протестирован на существующем наборе данных и показал высокое качество предсказания истинности предупреждений;
- механизм автоматической классификации предупреждений интегрирован в инфраструктуру статического анализатора **Svace** [30];
- реализовано отображение результатов классификации предупреждений в пользовательском интерфейсе с возможностью сортировки и фильтрации.

Список литературы / References

[1]. Tsiazhkorob U. V., Ignatyev V. N. Classification of Static Analyzer Warnings using Machine Learning Methods //2024 Ivannikov Memorial Workshop (IVMEM). – IEEE, 2024. – С. 69-74.

[2]. Иванников В. П. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ //Труды Института системного программирования РАН. – 2014. – Т. 26. – №. 1. – С. 231-250.
[3]. Lee S. et al. Classifying false positive static checker alarms in continuous integration using convolutional neural networks //2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). – IEEE, 2019. – С. 391-401.
[4]. Alikhashashneh E. A., Raje R. R., Hill J. H. Using machine learning techniques to classify and predict static code analysis tool warnings //2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA). – IEEE, 2018. – С. 1-8.
[5]. Christmann A., Steinwart I. Support vector machines. – 2008.
[6]. Dasarathy B. V. Nearest neighbor (NN) norms: NN pattern classification techniques //IEEE Computer Society Tutorial. – 1991.
[7]. Breiman L. Random forests //Machine learning. – 2001. – Т. 45. – С. 5-32.
[8]. Witten I. H. et al. Data Mining: Practical machine learning tools and techniques. – Elsevier, 2025.
[9]. Rajput A. et al. J48 and JRIP rules for e-governance data //International Journal of Computer Science and Security (IJCSS). – 2011. – Т. 5. – №. 2. – С. 201.
[10]. Understand: The Software Developer's Multi-Tool [Электронный ресурс]. – URL: https://scitools.com.
[11]. CWE – Common Weakness Enumeration [Электронный ресурс]. – URL: https://cwe.mitre.org.
[12]. Center for Assured Software N.S.A. Juliet Test Suite v1.1 for C/C++ User Guide. [Электронный ресурс]. – URL: https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.1_for_C_Cpp_-_User_Guide.pdf.
[13]. Software metric [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Software_metric
[14]. Lee M. C. Software quality factors and software quality metrics to enhance software quality assurance //British Journal of Applied Science & Technology. – 2014. – Т. 4. – №. 21. – С. 3069-3095.
[15]. Белеванцев А. А., Велесевич Е. А. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ //Труды Института системного программирования РАН. – 2015. – Т. 27. – №. 2. – С. 53-64.
[16]. McKinney W. et al. Data structures for statistical computing in Python //SciPy. – 2010. – Т. 445. – №. 1. – С. 51-56.
[17]. Pedregosa F. et al. Scikit-learn: Machine learning in Python //the Journal of machine Learning research. – 2011. – Т. 12. – С. 2825-2830.
[18]. Beautiful Soup: We called him Tortoise because he taught us. [Электронный ресурс]. URL: https://www.crummy.com/software/BeautifulSoup/.
[19]. Dorogush A. V., Ershov V., Gulin A. CatBoost: Gradient boosting with categorical features support. arXiv 2018 //arXiv preprint arXiv:1810.11363. – 1810.
[20]. Parmar A., Katariya R., Patel V. A review on random forest: An ensemble classifier //International conference on intelligent data communication technologies and internet of things. – Cham : Springer International Publishing, 2018. – С. 758-763.
[21]. Chen T., Guestrin C. Xgboost: A scalable tree boosting system //Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. – 2016. – С. 785-794.
[22]. Cohesion (computer science) [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/Cohesion_(computer_science).
[23]. Belevantsev A. et al. Design and development of Svace static analyzers //2018 Ivannikov Memorial Workshop (IVMEM). – IEEE, 2018. – С. 3-9.
[24]. T. Kremenek. Finding software bugs with the Clang static analyzer [Электронный ресурс]. URL: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf.
[25]. SpotBugs: Find bugs in Java Programs [Электронный ресурс]. URL: https://spotbugs.github.io.
[26]. Koshelev V. K. et al. SharpChecker: Static analysis tool for C# programs //Programming and Computer Software. – 2017. – Т. 43. – С. 268-276.
[27]. Svacer Wiki [Электронный ресурс]. URL: https://svacer.ispras.ru/.
[28]. О центре | Linux Verification Center. URL: http://linuxtesting.ru/about.
[29]. Swagger UI: Svacer Server 10.x.x. Svacer REST API documentation [Электронный ресурс]. URL: https://svacer-demo.ispras.ru/api/public/swagger/.
[30]. AI-ассистент для разметки предупреждений анализатора [Электронный ресурс]. URL: https://svace.pages.ispras.ru/svace-website/2025/02/21/ai-assistant.html.

Информация об авторах / Information about authors

Ульяна Владимировна ТЯЖКОРОБ – аспирант Физтех-школы Радиотехники и Компьютерных Технологий МФТИ, сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ, машинное обучение.

Uljana TSIAZHKAROB – postgraduate student of the Phystech School of Radio Engineering and Computer Technologies of MIPT, employee of the ISP RAS. Research interests: compiler technologies, static program analysis, machine learning.

Михаил Владимирович БЕЛЯЕВ – младший научный сотрудник ИСП РАН. Научные интересы: компиляторные технологии, статический анализ программ.

Mikchail Vladimirovich BELYAEV – researcher at ISP RAS. Research interests: compiler technologies, static program analysis.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор кафедры системного программирования ВМК МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., leading researcher at ISP RAS, Professor at Moscow State University. Research interests: static analysis, program optimization, parallel programming.

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

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

DOI: 10.15514/ISPRAS-2025-37(6)-7

Fast Calls and In-Place Expansion: A Hybrid Strategy for VM Intrinsics

D.V. Zavedeev, ORCID: 0009-0009-1477-5249 <zdenis@ispras.ru>
R.A. Zhuykov, 0000-0002-0906-8146 <zhroma@ispras.ru>
L.V. Skvortsov, ORCID: 0000-0002-1580-1244 <lvs@ispras.ru>
M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This research proposes a hybrid approach for implementing performance-oriented compiler intrinsics. Compiler intrinsics are special functions that provide low-level functionality and performance improvements in high-level languages. Current implementations typically use either in-place expansion or call-based methods. In-place expansion can create excessive code size and increase compile time but it can produce more efficient code in terms of execution time. Call-based approaches can lose at performance due to call instruction overhead but win at compilation time and code size. We survey intrinsics implementation in several modern virtual machine compilers: HotSpot Java Virtual Machine, and Android RunTime. We implement our hybrid approach in the LLVM-based compiler of Ark VM. Ark VM is an experimental bytecode virtual machine with garbage collection and dynamic and static compilation. We evaluate our approach against in-place expansion and call approaches using a large set of benchmarks. Results show the hybrid approach provides considerable performance improvements. For string-related benchmarks, the hybrid approach is 6.8% faster compared to the no-inlining baseline. Pure in-place expansion achieves only 0.7% execution time improvement of the hybrid implementation. We explore two versions of our hybrid approach. The "untouched" version lets LLVM control inlining decisions. The "heuristic" approach was developed after we observed LLVM's tendency to inline code too aggressively. This research helps compiler developers balance execution speed with reasonable code size and compile time when implementing intrinsics.

Keywords: intrinsics; LLVM compiler infrastructure; virtual machines.

For citation: Zavedeev D.V., Zhuykov R.A., Skvortsov L.V., Pantilimonov M.V. Fast Calls and In-Place Expansion: A Hybrid Strategy for VM Intrinsics. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 121-134. DOI: 10.15514/ISPRAS-2025-37(6)-7.

Быстрые вызовы и раскрытие на месте: гибридная стратегия для встраиваемых функций виртуальных машин

Д.В. Заведеев, ORCID: 0009-0009-1477-5249 <zdenis@ispras.ru>
Р.А. Жуйков, 0000-0002-0906-8146 <zhroma@ispras.ru>
Л.В. Скворцов, 0000-0002-1580-1244 <lvs@ispras.ru>
М.В. Пантилимонов, 0000-0003-2277-7155 <pantlimon@ispras.ru>

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

Аннотация. Данное исследование предлагает гибридный подход к реализации встраиваемых компиляторных функций, направленных на улучшение производительности. Встраиваемые компиляторные функции – особые функции, которые предоставляют доступ к низкоуровневым возможностям или улучшают производительность. Текущие реализации, как правило, используют либо раскрытие на месте, либо подходы, основанные на вызове. Раскрытие на месте может избыточно увеличить размер кода и время компиляции, но создать более эффективный код по времени исполнения. Подходы, основанные на вызове, могут проигрывать по производительности в связи с вызовом функции, но выигрывают по размеру кода и времени компиляции. Мы рассматриваем реализации встраиваемых функций в нескольких современных компиляторах виртуальных машин: в виртуальной машине Java HotSpot и в Android RunTime. Мы реализуем гибридный подход в LLVM-компиляторе для виртуальной машины Ark. Ark – это экспериментальная байткодная виртуальная машина со сборкой мусора, динамическим и статическим компиляторами. Мы сравниваем наш гибридный подход с раскрытием на месте и подходом, основанном на вызовах, на большом наборе бенчмарков. Результаты показывают, что гибридный подход по времени исполнения показывает значительное улучшение. Строковые бенчмарки выполняются быстрее на 6.8% по сравнению с подходом, основанном исключительно на вызовах, в то же время, чистое раскрытие на месте быстрее на 0.7% гибридного подхода. Мы рассматриваем две версии гибридного подхода. "Untouched" версия позволяет LLVM самому принимать решение о встраивании функции или выборе вызова. Подход "heuristic" мы разработали после того, как заметили, что LLVM в "untouched" подходе производит излишне агрессивное встраивание функций. Данная статья поможет разработчикам компиляторов найти баланс между временем исполнения, размером кода и временем компиляции при реализации встраиваемых функций.

Ключевые слова: встраиваемые функции; компиляторная инфраструктура LLVM; виртуальные машины.

Для цитирования: Заведеев Д.В., Жуйков Р.А., Скворцов Л.В., Пантилимонов М.В. Быстрые вызовы и раскрытие на месте: гибридная стратегия для встраиваемых функций виртуальных машин. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 121–134 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–7.

1. Introduction

Intrinsics are functions that compilers handle in a special way. They serve two primary purposes: first, providing programmers access to low-level functionality within high-level languages like C or C++ (such as the `__rdtsc` function, which returns cycle count for a CPU) and second, enhancing application performance (such as optimized implementations of `Math.log` in HotSpot JVM [1]). This paper focuses specifically on performance-oriented intrinsics.

Compilers typically implement intrinsics using two main strategies: in-place expansion, where the intrinsic call is expanded into a sequence of machine instructions directly at the call site; and the call approach, which generates a call to precompiled or runtime-generated code. Our survey shows that modern compilers utilize both approaches, though implementation practices vary across different compiler systems.

Virtual machine (VM) compilers often expand intrinsics without code size limitations, which can lead to excessive memory usage and potential performance degradation. Conversely, some VM compilers never use in-place expansion, instead relying exclusively on efficient calls. While this approach preserves memory and reduces code size, it might introduce performance penalties. On the other side, Damásio et al. propose using an inlining to reduce code size [2].

In this paper, we propose a hybrid approach that combines both in-place expansion and efficient calls. This method aims to deliver performance benefits while maintaining reasonable code size and compile time.

We implement the hybrid approach and evaluate its performance within the LLVM-based [3] compiler of Ark Virtual Machine (Ark VM) [4]. Ark VM is an experimental bytecode virtual machine with garbage collection, dynamic (Just in Time – *JiT*) and static (Ahead of Time – *AoT*) compilers. It executes bytecode compiled from ETS, a statically typed language similar to TypeScript (but other frontends are possible to the VM's bytecode).

The hybrid approach has experimental status, and shows promising results: for string-related benchmarks execution time is 0.932 (6.8%) times of the no-inlining baseline, while the always *in-place* expansion is only 0.993 (0.7%) times of the *hybrid* approach performance.

We have been working on LLVM-based compilation for Ark VM, thus we can experiment with intrinsics optimizations with relative ease in the LLVM-based compiler.

In this paper, we begin by introducing Ark VM in the 2nd section, where its essential components are described. Next, we survey various ways to encode intrinsics in the HotSpot Java Virtual Machine [5] and Android RunTime [6], while also including Ark VM for comparison in the 3rd section. The 4th section then presents a hybrid approach that combines both *call* and *in-place* expansion techniques.

Following this, the 5th section compares the *in-place*, *calls*, and *hybrid* approaches to intrinsics encoding in Ark VM, focusing on code size, application performance, and compile time. To provide empirical evidence, we run a comprehensive set of benchmarks, demonstrating how performance varies across a wide range of scenarios. Additionally, a dedicated micro-benchmark for the `StringEquals` intrinsic is developed, allowing us to analyze in detail the advantages and disadvantages of each approach.

We also discuss two flavors of the hybrid approach: *untouched* and *heuristic* because our setup reveals that LLVM tends to be overly aggressive with inlining, as will be shown in the 5.4 section. The paper concludes by summarizing the results.

2. Ark VM

2.1 Execution in Ark VM

Ark Virtual Machine is a bytecode virtual machine.

Compilers like Clang or GCC transform programs in a high-level language like C into a sequence of machine instructions specific to the CPU. In this approach a program compiled for x86_64 CPU cannot be run on aarch64 CPU directly.

We illustrate it in Fig. 1: compiler takes a program in C as an input, and produces an executable file, which in turn runs on x86_64 CPU.

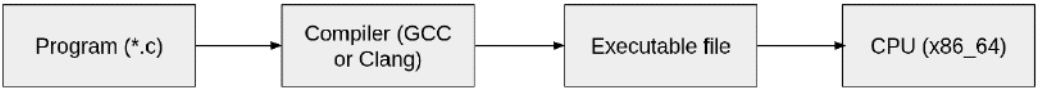


Fig. 1. Compilation and execution for compilers producing directly executable files.

For VMs like Ark it is different: a compiler for high-level language transforms a program into a sequence of *bytecode* instructions. *Bytecode* is a set of instructions for *artificial* processor that does not exist but is a convenient abstraction. The VM emulates such a processor. Thus, a bytecode program runs anywhere where the VM is available, despite environment.

Fig. 2 is specific to Ark VM: ETS is a programming language, ETS frontend is a program that transforms ETS to bytecode, it takes a program in ETS language, and produces bytecode file (`.abc`), which then runs on Ark VM.

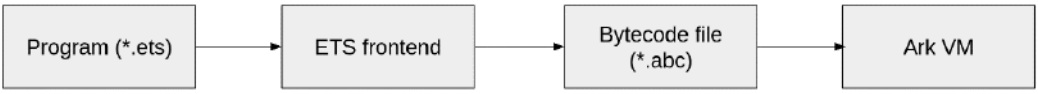


Fig. 2. Compilation from high-level language to bytecode file and execution in Ark VM.

Fig. 1 and Fig. 2 look similar but there is a huge difference: in the first case, the executable file runs directly on CPU, in the second case Ark VM acts as an emulator for CPU (Ark VM is a program itself). To emulate a CPU VM has an *interpreter* – a component which reads instructions from bytecode file one by one and emulates their behavior. Such an interpreter in its simplest form is implemented in the language of VM, e.g., C++, using a dispatch table. Modern VMs employ different optimizations techniques to make interpretation faster, e.g., HotSpot JVM has a template interpreter generated at runtime [7].

Despite improvements made to the interpreter, interpretation remains slow. To alleviate this, VMs employ compilers. VM monitors itself during the execution, and for frequently executed functions *Just in Time* compiler produces native code for the CPU where the VM runs at *runtime*. VMs also employ *Ahead of Time* compilers which produce native code *before* the execution. *AoT* compiler is used in different scenarios when upfront compilation is more profitable: e.g., to compile parts of the standard library, or parts of an application. Ark VM has both *JiT* and *AoT* compilers.

2.2 Ahead of Time compilation in Ark VM

AoT compiler in Ark VM transforms bytecode file into a file with instructions executable by CPU directly.

In Fig. 3 we show the inputs for Ark VM: Ark VM requires a file with bytecode for execution, and optionally AoT compiled files – `*.an` with machine code. Ark VM upon start loads the AoT compiled files, and registers the machine code internally: for each AoT compiled function, the VM changes its entry point so that instead of interpretation, the AoT compiled machine code is executed.

If there is no registered machine code for a function, then VM starts with interpretation of the function. If the function is executed often enough, *JiT* compiler produces native code dynamically.

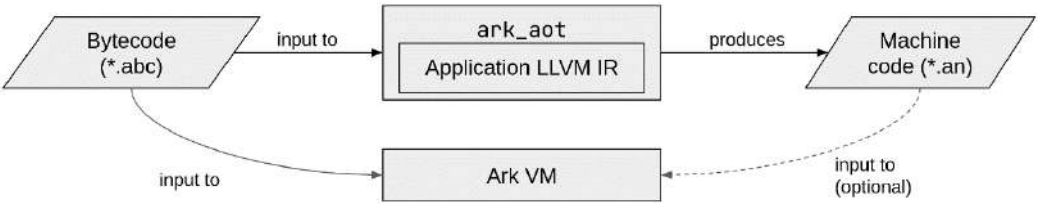


Fig. 3. Inputs into Ark VM: bytecode file (mandatory), AoT-file (optional).

In Ark VM there are two kinds of *AoT* compilers: stock, and LLVM-based. Stock compiler is written from scratch for the VM, and LLVM-based compiler transforms bytecode into LLVM IR, then LLVM produces native code for that LLVM IR. In this paper we focus only on LLVM-based AoT compiler.

2.3 Intrinsics in Ark VM

In Ark VM there are two kinds of intrinsics:

1. Regular intrinsics which are encoded using various strategies. E.g., to get a length of a String compiler uses special Intermediate Representation (IR – form of code between the source language and machine code). Also, there are intrinsics which are replaced by calls to the standard library function calls like `sin`.
2. FastPaths – precompiled intrinsics or VM routines.

2.4 FastPaths in Ark VM

To call a FastPath compiler produces a *fast-call* in Ark VM.

At the low-level, to call a function the compiler must obey a set of rules which define how to pass arguments to a function, how to return a value from a function, which registers must be preserved across calls (callee-saved registers), and are not guaranteed to be preserved (caller-saved registers or call-clobbered registers). This set of rules is *calling convention* in compiler terminology.

Fast-call means using a special calling convention which preserves almost all registers across calls. It uses `ArkFast` calling convention in our patched LLVM 15 version.

The purpose of the calling convention is to make the code for caller as cheap as possible, so the caller does not need to preserve registers before making a call. The `ArkFast` calling convention is supported for aarch64 and x86_64. The calling convention has also caller saved registers: 4 for aarch64, and 6 for x86_64. Parameters, and return value registers are caller or callee-saved depending on their presence. For comparison, the default calling convention for arm64 – aapcs64 defines more than 20 of the registers as call-clobbered [8].

2.5 Intrinsic FastPaths example: StringEquals

Consider the following code in ETS language in Listing 1.

```
function bar(a: string, b: string): boolean {
    return a === b;
}
```

Listing 1. String comparison in ETS language.

The code in the Listing 1 uses the `'==='` operator to check if two strings are equal. Two strings are equal if their lengths are equal and characters composing these strings are equal. In Ark VM the performance of such a comparison is so critical that string comparison is an intrinsic function – `StringEquals`.

`StringEquals` intrinsic in AoT code is encoded as a fast-call to precompiled machine code.

2.6 Categories of FastPaths

Ark VM has 90 FastPaths. There are FastPaths for high-level operations like comparing Strings which directly map onto standard library methods. Also, there are low-level FastPaths, e.g., allocation routines. We categorize all FastPaths in Table 1.

As we can see, most of the FastPaths are related to strings in Ark VM.

2.7 Summary

1. VM emulates artificial CPU to make programs for the VM cross-platform. Instructions for such cpu are *bytecode* instructions. Ark VM is one of such machines.

2. When executing the bytecode Ark VM starts with interpretation, then *JiT* compiler can produce machine code if the function is frequently executed, *AoT* compiler can produce machine code before execution.
3. There are two kinds of intrinsics in Ark VM: regular and FastPath. Regular intrinsics are encoded using various strategies. FastPath is precompiled machine code for VM routines or intrinsics. Most of the FastPaths are string related in Ark VM.

Table 1. Categories of FastPaths in Ark VM.

Category	Count	Comment
Allocation	5	Allocation routines for objects and arrays
Garbage Collection (GC)	9	GC pre-, post-write barriers
String	61	String routines: creating a substring, appending to string, etc.
Array copy	11	Various routines to support ArrayCopy operation in ETS
Misc	4	Interface lookup cache, monitor lock and unlock, check cast

3. Industry approaches to intrinsics

In this section we survey approaches to intrinsics encoding. We do it for HotSpot JVM (openjdk version "21.0.1" 2023-10-17), Android RunTime (specifically dex2oat aml_art_351110180, Nov 2024), and Ark VM.

We observe the following strategies of intrinsics encoding in production VMs:

1. *Call replacement*. VM compiler replaces a call to intrinsic function with a call to another equivalent function but more performant. Android RunTime, Ark, HotSpot JVM perform call replacement for sin function: Android RunTime [9] and Ark replace it with a call to the `std::sin` function, HotSpot JVM replaces it with a call to the runtime generated `_dsin` function [10].
2. *Special IR*. In this case, compilers replace a call with a special sequence of IR instructions. Example of such strategy is encoding of `java.lang.String::length` in Android RunTime [11]. Ark also replaces calls to get string length with several compiler IR instructions, these instructions become a part of the function's body, and are optimized together.
3. *In-place expansion*. In this case, compiler replaces a call to an intrinsic function with a sequence of machine instructions at the call site. We observed that behavior in HotSpot JVM for `java.lang.Arrays::equals(byte[], byte[])` method [12]. Android RunTime also employs *in-place* expansion for `java.lang.String::equals` method [13].
4. *Fast-call*. A special case for call replacement. Ark VM employs this approach for FastPath intrinsics. As we described in the 2.4 section, in this case a call to an intrinsic is encoded as a call with calling convention that preserves almost all registers.
5. *Calling other intrinsics*. OpenJDK 21 employs this approach to improve the performance of `String::equals` method: in OpenJDK 21, the method is not intrinsic itself, but it calls `StringLatin1::equals` intrinsic. Júnior Löff et al. [14] propose leveraging that method, and suggest replacing several intrinsics written in assembly with implementation in Java using Java Vector API.

Each of the approaches has its advantages and disadvantages. *Call replacement*, *in-place expansion*, and *fast calls* are used to implement complex high-level operations, like comparing strings. *Special*

IR is usually used for simple operations like getting a string length. *Fast-call* is similar to *call replacement* since those approaches are basically calls.

We are interested in the *in-place expansion* and *call* approaches because using solely either of approaches has its own advantages and disadvantages. We list them in Table 2.

We see that HotSpot JVM, Android RunTime, and Ark VM lack flexibility in encoding intrinsics: for some of the intrinsics their approach is *black-or-white*, with no option to adapt or vary the encoding method.

We propose an approach that allows the encoding strategy to change based on potential benefits.

Our hybrid approach aims to be the approach taking the best of the *in-place* and the *call* approaches: reasonable code size, improved performance, and manageable compile time.

Table 2. In-place and call approaches compared.

Criteria	**In-place**	**Call**
Code size	Usually grows indefinitely	No code size growth
Performance	Can produce more performant code due to (1) absence of a call instruction, (2) possible optimizations after inlining	Can lose at performance
Compile time	Takes more time to compile	Less time to compile

4. Hybrid approach to intrinsics encoding in Ark VM

4.1 VM before introducing hybrid approach

As of current state, Ark VM produces precompiled machine code for FastPaths – VM routines and high-level operations like comparing strings. We illustrate it in Fig. 4: FastPath compiler produces a `fastpath.o` file with machine code from the source code, then the `fastpath.o` file becomes part of the runtime library.

In Fig. 5 we show how AoT compiler transforms bytecode file into a file with machine code. When compiling calls to FastPaths, compiler produces calls to functions from the dynamic library (`libarkruntime.so`) – a part of VM runtime. AoT compiler has no definitions of the FastPath functions, and thus cannot inline the FastPath into the application code.

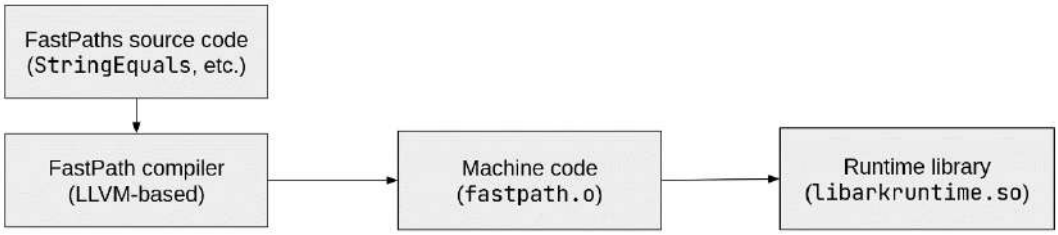


Fig. 4. FastPath compilation before hybrid approach.

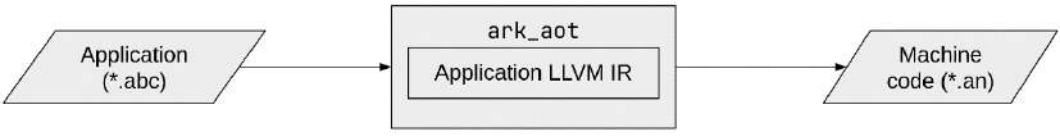


Fig. 5. AoT compilation before hybrid approach.

4.2 VM with hybrid intrinsics encoding

We involve joint work of two compilation phases. First, during VM build, LLVM compiles FastPaths and dumps LLVM bitcode. We show the difference in Fig. 6: now we produce not only a binary file with machine code (`.o`) but also a bitcode file with definitions of the FastPaths (marked blue in the diagram).

Second, during AoT compilation we supply the dumped bitcode file for joint optimizations purposes. Fig.7 now shows that *AoT* compiler has LLVM IR from bytecode, and for FastPath.

Now, LLVM *AoT* compiler has the definitions of the FastPath functions, not only their declarations, and can inline the FastPath functions into *AoT*-code.

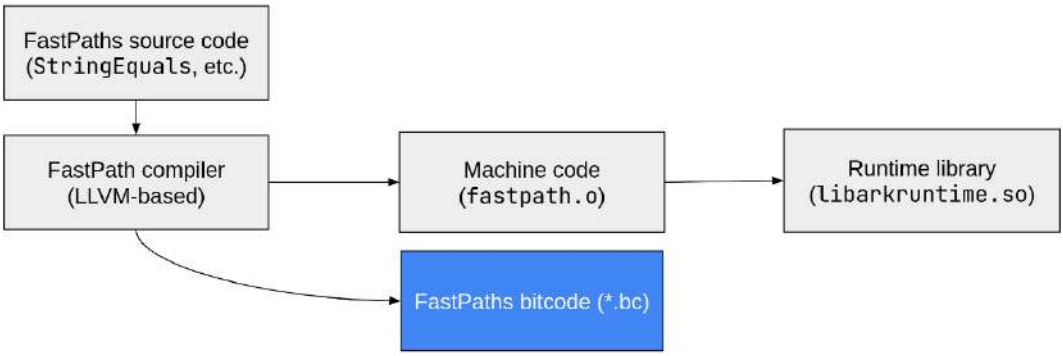


Fig. 6. FastPath compilation with hybrid approach.

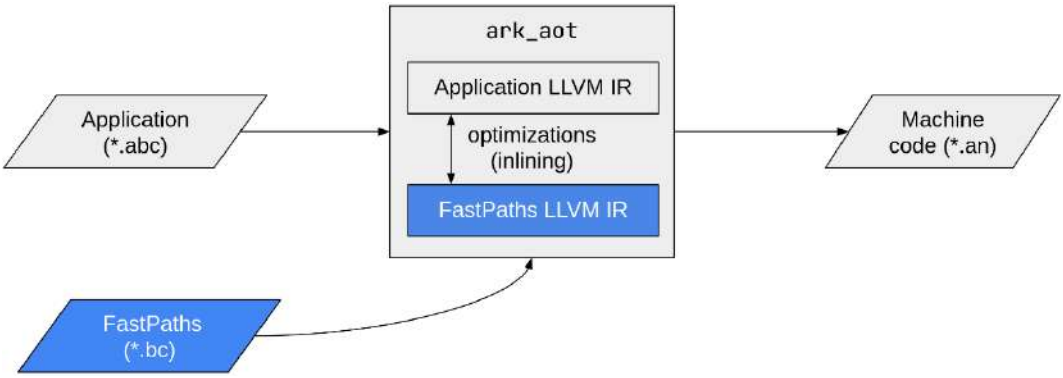


Fig. 7. AoT compilation with hybrid approach.

New parts are blue in these diagrams. All grey components in the diagrams are already implemented. VM uses the produced `.an` file as usual when started: it loads the `.an` file upon start, registers compiled machine code, and executes more optimal compiled code.

4.3 Controlling inlining

To allow fine-tuning of inlining we introduce an LLVM pass which forces or forbids inlining of FastPaths. `fastpath-max-always-inlines` option controls the number of forcefully inlined FastPaths into AoT compiled function:

- N > 0 – forces inlining of the first N FastPaths met in the function, forbids inlining of other FastPaths in function
- 0 – forbids inlining of all FastPaths in a function – equivalent of default mode
- -1 – neither force, nor forbid inlining of FastPaths. In this case, the decision is up to LLVM.

A FastPath can call another FastPath in Ark VM, in this case, we inline the FastPath unconditionally.

5. Evaluation

In this section we evaluate configurations of FastPath inlining listed in Table 3.

We introduce the "heuristic" scenario in our measurements because we know that LLVM is too aggressive at inlining in advance, as it will be evidenced in the 5.4 section.

Table 3. Inlining configurations.

Approach	Description	fastpath-max-always-inlines
No inlining	Inlining of all FastPaths into AoT code is forbidden	0
Always inline	Inlining of all FastPaths into AoT code is enforced	2 000 000 000
Heuristic	Only the first occurrence of FastPath in the function is inlined forcefully, inlining is forbidden for other FastPaths in the function	1
Untouched	Leave the decision up to LLVM	-1

We are interested in three metrics:

1. Execution time – how long it takes to perform an operation.
2. Code size – the size of a produced binary AoT file. In Ark these are files with `.an` extension.
3. Compilation time – how long it takes to compile a file.

To measure code size, and compilation time we compile the standard library file of ETS language. To measure performance, we run a large set of benchmarks. We also gather compile time and code size for benchmark files.

But first, we describe which FastPaths we inline and do not, and describe our setup.

In our evaluations we do not inline:

1. GC-related FastPaths because they are called by a pointer, thus it is impossible to know which FastPath is called.
2. "Interposer" FastPaths. They simply wrap a single function call, and never called by AoT code. There are 2 interposer FastPaths.

That is, we inline 78 FastPaths out of 90.

Another important note: to provide a fair comparison between approaches, we disabled safepoints in our experimental code. Safepoints function like scheduled checkpoints where running code briefly pauses to allow the virtual machine to perform maintenance tasks such as garbage collection. When FastPath code is compiled on its own, the compiler does not insert safepoints because they are not needed in this specialized code. However, when we inline this FastPath code into the main program, the compiler automatically inserts safepoint instructions into loops, treating it like regular code.

This automatic insertion creates an inconsistency: the same FastPath code performs differently when inlined versus when kept separate, not because of our hybrid approach but because of how the compiler handles it. By disabling safepoints throughout all code paths, we establish a consistent environment that allows us to measure the true performance differences between our approaches without this compiler introduced variable affecting our results.

While safepoints are important in production environments, disabling them for our comparative analysis ensures our research findings reflect the actual differences between implementation approaches rather than artifacts of the compilation process.

5.1 Setup

We run all measurements:

- On aarch64 Linux.

- Using Release built version of Ark VM and Release LLVM.
- Using vm-benchmarks framework [15].

5.2 Standard library: code size, and compile time

We compile `etsstdlib` – the bytecode of ETS standard library, resulting file with machine code is `etsstdlib.an`. The file has 10764 calls to FastPaths. Table 4 summarizes our measurements. In parentheses, we show score against the "No-inlining baseline".

Table 4. Different inlining strategies for standard library.

Approach	Size, bytes	Compile time, s
Always inline	7 489 992 (x1.39)	303 (x1.62)
No inlining	5 384 648 (x1)	187 (x1)
Heuristic	5 642 696 (x1.05)	200 (x1.07)
Untouched	6 101 448 (x1.13)	220 (x1.18)

As we could expect, the "Always inline" approach grows the size of binary file the most (x1.39 of original), compile time is also at the maximum. The "Untouched" approach grows the code size by x1.13 of the original, compile time is also lower than in "Always inline". Our heuristic approach provides minimal code size growth of the binary file, and minimal compile time growth.

5.3 Benchmarks: performance, compile time, and code size

We run a large set of benchmarks. As we described in the section 2.6, most of the FastPaths are string related, so we should keep an eye on benchmarks related to strings. We evaluate our approach on 2305 benchmarks. Those include 276 String benchmarks.

We show relative value which should be read as "x times of no inline baseline".

The relative values are geometric mean ratios across metrics: execution time, `.an`-file size, and compile time. For all cases "lower is better".

In our measurements, the VM uses AoT-compiled standard library. That is, if a benchmark calls a function from the standard library, then the called function is also *AoT*-compiled with `fastpath-max-always-inlines` set to match the approach under measurement.

5.3.1 String benchmarks

For string benchmarks the results are in Table 5. As we can see, the always inline approach has the best execution time: 0.925 of the no-inlining baseline, the worst compile time, and code size – 2.381, 1.313 of the original respectively.

The untouched approach is 0.957 of the original at execution time. Code size is almost the same as the original – 1.015. Compile time is less than for "always inline" approach.

The heuristic approach is closer in performance to the always inline approach, and does not increase compile and code size dramatically.

Table 5. String benchmarks results

Characteristic\Approach	Always inline	Heuristic	Untouched
Execution time	0.925	0.932	0.957
Code size	1.313	1.059	1.015
Compile time	2.381	1.302	1.250

5.3.2 All benchmarks

For the whole set of benchmarks, the results are in Table 6. In the whole set of benchmarks overall performance gains drop compared to the string benchmarks.

This happens because of the nature of FastPath we are inlining: most of them are String related.

Table 6. All benchmarks results.

Characteristic\Approach	Always inline	Heuristic	Untouched
Execution time	0.980	0.999	0.998
Code size	1.220	1.044	1.037
Compile time	1.879	1.159	1.192

5.4 StringEquals benchmark

In this section we take benchmarks to extreme: usually a benchmark has a reasonable number of intrinsic calls, we are going to vary a number of calls from 50 to 300 and see results.

Mostly, this section motivates using the heuristic approach, because we see the evidence that leaving inlining decision entirely up to LLVM can increase code size significantly, and degrade performance. The *heuristic* approach solves this problem: it forces inlining of the first FastPath call met in a function, and disables inlining of all other FastPaths in the function. We have not tried sophisticated heuristics since in our opinion this research must be done against the production ready implementation of FastPath inlining, and as of now safepoint support is missing. Despite that our guess to inline the first FastPath call is quite good in all terms, which we are interested in.

We run benchmark, where we call `String::equals` ranging number of times between 50 and 300 for 4 and 1024 length strings. Strings have equal lengths, and differ only in the last character.

Fig. 8 visualizes the results for 4 length strings. For 1024 string length graph with performance is in Fig. 9.

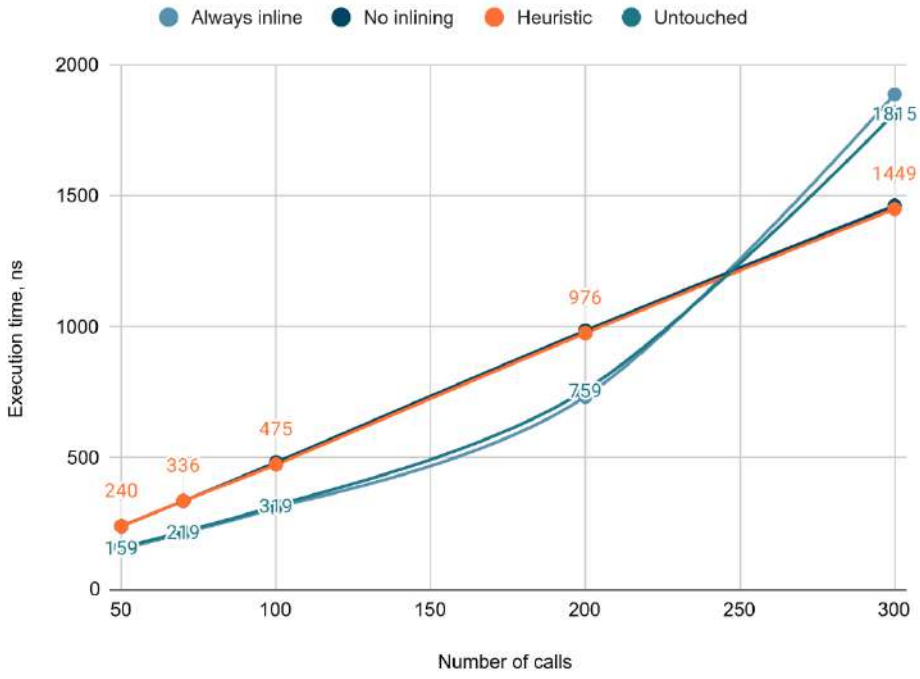


Fig. 8. Performance for 4 characters string.

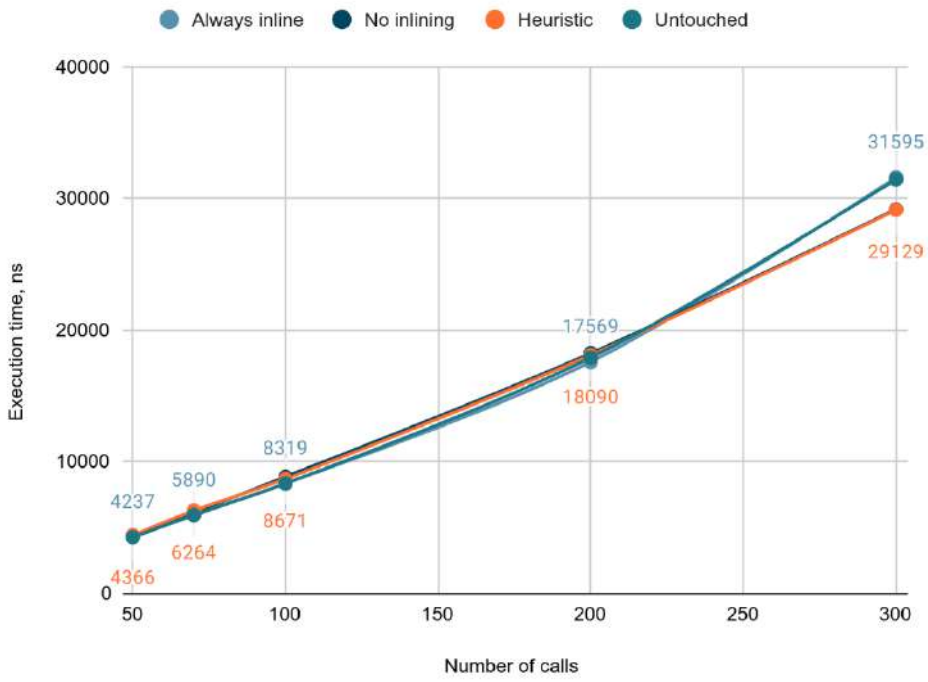


Fig. 9. Performance for 1024 characters string.

There are interesting points to note:

- *Untouched* and *always inline* approaches show similar performance
- *Untouched* and *always inline* approaches have lower performance for 300 number of calls compared to the *heuristic* and to the *no-inlining* approaches
- For 1024 length strings there is no significant difference between approaches since the strings are long, comparing 1024 characters outweighs eliminated call overhead.

The reason why *untouched* approach shows performance close to *always inline* is that when calls are left *untouched*, LLVM inlines almost all the `StringEquals` calls.

For 300 number of `StringEquals` calls the *no-inlining* and *heuristic* cases are more performant, independently of the string length. The reason we *think* of is that the machine code in *always inline* and *untouched* cases does not fit the instruction cache. A CPU we use has 64KiB of instruction cache per core, code size is 106.71KiB in *always inline* case, which is larger than the CPU cache.

As we see, leaving the decision about inlining up to LLVM can degrade performance and increase code size.

Because of increased code size, and possible performance degradation we have introduced the *hybrid* approach in favor of the *untouched* approach.

6. Future work

We see our future work as follows:

1. Safepoint support. We had to disable safepoint placement to make comparison fair but to make the *hybrid* approach ready for production, we should allow safepoint placement without sacrificing the performance of FastPaths.

2. More intrinsics. We based our work only on a part of intrinsics that are FastPaths. Other intrinsics can be migrated to FastPath so that we could inline them too.
3. x86_64 support. For now, hybrid approach supports only aarch64 architecture, though LLVM AoT compiler supports x86_64 and aarch64.
4. Heuristic improvement. Our heuristic to always inline the first FastPath met in the function already shows decent performance, compile time, and code size but still there can be more effective heuristics.

7. Conclusion

We proposed a hybrid approach to intrinsics encoding for VMs' compilers. We implemented the approach in LLVM-based static compiler for Ark VM. Our approach combines two approaches: *fast-calls* and *in-place expansion*.

We compared the hybrid approach to pure *fast-calls* and *in-place* expansion. We present a hybrid approach in two flavors: *untouched* where LLVM makes an inlining decision based on its own metrics, and *heuristic* which forces inlining of the first intrinsic met in a function and forbids inlining of other intrinsics.

The *heuristic* flavor shows better performance, code size management, and compile time compared to the *untouched* version. For string-related benchmarks, the hybrid heuristic approach is 6.8% faster compared to the no-inlining baseline while the pure in-place expansion achieves only 0.7% execution time improvement of the hybrid heuristic implementation.

Intermediate language for intrinsics like LLVM IR gives more control over intrinsics encoding: we can inline them, leave them as calls, or leave the decision up to compiler.

References

[1]. Dehghani A. HotSpot Intrinsics. https://alidg.me/blog/2020/12/10/hotspot-intrinsics (accessed May 14, 2025).
[2]. Damásio T., Pacheco V., Goes F., Pereira F., and Rocha R. Inlining for Code Size Reduction. In Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP '21), Association for Computing Machinery, New York, NY, USA, 2021. doi: 10.1145/3475061.3475081
[3]. The LLVM Project. https://llvm.org/ (accessed May 20, 2025).
[4]. ArkCompiler Runtime Core: Static Core. https://gitee.com/openharmony/arkcompiler_runtime_core/tree/master/static_core (accessed May 14, 2025).
[5]. Oracle. Java Virtual Machine Technology Overview. https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html (accessed May 20, 2025).
[6]. Android Open Source Project. Android runtime and Dalvik. https://source.android.com/docs/core/runtime (accessed May 20, 2025).
[7]. HotSpot Runtime Overview: Interpreter. https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html#Interpreter|outline (accessed May 14, 2025).
[8]. ARM Software. Procedure Call Standard for the Arm® 64-bit Architecture (AArch64): The Base Procedure Call Standard. https://github.com/ARM-software/abi-aa/blob/main/aapcs64/aapcs64.rst#the-base-procedure-call-standard (accessed May 14, 2025).
[9]. Android RunTime (ART) ARM64 sin Entrypoint Initialization. https://cs.android.com/android/platform/superproject/main/+/main:art/runtime/arch/arm64/entrypoints_init_arm64.cc;l=187;drc=0e8091312485670e84ee17daf25256e5836112b0 (accessed May 14, 2025).
[10]. HotSpot Stub Generator for x86_64 sin Function. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/cpu/x86/stubGenerator_x86_64_sin.cpp (accessed May 14, 2025).
[11]. Android RunTime (ART) Compiler. HArrayLength instruction https://cs.android.com/android/platform/superproject/main/+/main:art/compiler/optimizing/nodes.h;l=6463;drc=621d1350d431ed0cc3d4a5a43a079adc1d86a31f (accessed May 14, 2025).

[12]. HotSpot C2 MacroAssembler for x86: Arrays Equals. https://github.com/openjdk/jdk/blob/60a4594b9f9acd82ef3ff22fc6a2df238dd981b9/src/hotspot/cpu/x86/c2_MacroAssembler_x86.cpp#L4377 (accessed May 14, 2025).

[13]. Android RunTime (ART) ARM64 Intrinsics. StringEquals encoding. https://cs.android.com/android/platform/superproject/main/+/main:art/compiler/optimizing/intrinsics_arm64.cc;l=2288;drc=9c2a0a8bfc5369a110956ac26cf9cf145a6a4bb7 (accessed May 14, 2025).

[14]. Löff J., Schiavio F., Rosà A., Basso M., and Binder W. Vectorized Intrinsics Can Be Replaced with Pure Java Code without Impairing Steady-State Performance. In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering (ICPE '24), Association for Computing Machinery, New York, NY, USA, 2024, pp. 14–24. doi: 10.1145/3629526.3645051

[15]. ArkCompiler Runtime Core: VM Benchmarks. https://gitee.com/openharmony/arkcompiler_runtime_core/tree/OpenHarmony_feature_20241108/static_core/tests/vm-benchmarks (accessed May 14, 2025).

Information about authors

Денис Владиславович ЗАВЕДЕЕВ – аспирант Института системного программирования им. В.П. Иванникова Российской академии наук. Сфера научных интересов: компиляторы, языковые виртуальные машины.

Denis Vladislavovich ZAVEDEEV – postgraduate student at Ivannikov Institute for System Programming of the Russian Academy of Sciences. Research interests: compilers, language virtual machines.

Роман Александрович ЖУЙКОВ – старший научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статическая и динамическая оптимизация программ, компиляторные технологии.

Roman Aleksandrovich ZHUYKOV – Senior researcher in Compiler Technology department at ISP RAS. Research interests: static and dynamic program optimization, compiler technologies.

Леонид Владленович СКВОРЦОВ – стажёр-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Leonid Vladlenovich SKVORTSOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

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

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

DOI: 10.15514/ISPRAS-2025-37(6)-8

Аннотирование исходного кода для статического анализа

[1] *В.О. Афанасьев, ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *А.Е. Бородин, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1] *Е.А. Велесевич, ORCID: 0009-0007-7850-6860 <evel@ispras.ru>*
[1,2] *Б.В. Орлов, ORCID: 0009-0009-1602-6014 <boris.orlov@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. В статье описывается аннотирование исходного кода для статического анализа. Рассмотрены атрибуты C/C++ и аннотации JVM-языков. Приведены основные цели и причины аннотирования исходного кода для статического анализа. Описаны основные аспекты реализации поддержки пользовательских аннотаций в анализаторе Svace.

Ключевые слова: статический анализ; поиск ошибок; уязвимости; анализатор Svace; компиляторная инфраструктура LLVM; виртуальная машина Java; языки программирования C/C++, Java, Kotlin.

Для цитирования: Афанасьев В.О., Бородин А.Е., Велесевич Е.А., Орлов Б.В. Аннотирование исходного кода для статического анализа. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 135–148. DOI: 10.15514/ISPRAS–2025–37(6)–8.

Source Code Annotation for Static Analysis

[1] *V.O. Afanasyev, ORCID: 0000-0002-8036-0633 <vafanasiev@ispras.ru>*
[1] *A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>*
[1] *E.A. Velesevich, ORCID: 0009-0007-7850-6860 <evel@ispras.ru>*
[1,2] *B.V. Orlov, ORCID: 0009-0009-1602-6014 <boris.orlov@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. This paper describes source code annotations for static analysis. C/C++ attributes and JVM annotations are considered. Primary goals and reasons for source code annotation for static analysis are given. Main implementation aspects of annotations in Svace static analyzer are described.

Keywords: static analysis; search for defects; vulnerabilities; Svace; LLVM; JVM; C/C++; Java; Kotlin.

For citation: Afanasyev V.O., Borodin A.E., Velesevich E.A., Orlov B.V. Source Code Annotation for Static Analysis. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 135-148' (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-8.

1. Введение

Инструменты статического анализа позволяют находить ошибки в исходном коде программ без их запуска и, во многих случаях, без предварительной подготовки программы. Тем не менее, не все свойства программ могут быть выведены автоматически. Для таких ситуаций решением будет добавление подсказок для анализатора об особенностях исходного кода.

Во многих языках есть встроенные средства по предоставлению дополнительной семантики. Например, язык Java предлагает аннотации к исходному коду, которые добавляют семантику программы. Пример такой аннотации – `Nullable`, которая подсказывает, что переменная может иметь нулевое значение.

Аннотации находятся вместе с кодом и не будут потеряны при рефакторинге, либо другом изменении кода. Также они видны программисту, который может поправить их. Поэтому логично будет использовать встроенные средства языка для подсказок статическому анализатору.

В данной работе мы описываем использование пользовательских аннотаций для языков C, C++, Java и Kotlin для улучшения статического анализа программ. Все описанные работы выполнены в инструменте Svace [1, 2]. Аннотации добавляют информацию для анализатора, при этом не являются обязательными. В разделе 2 описываются возможности аннотирования кода в языках C, C++, Java, Kotlin. В разделе 3 приводятся возможные способы использования аннотаций для статических анализаторов. Реализация в инструменте Svace находится в разделе 4. Результаты тестирования приведены в разделе 5.

2. Способы аннотирования кода в языках программирования

2.1 Атрибуты C/C++

Атрибуты – устоявшийся способ передачи компилятору C/C++ дополнительной информации. Так, существуют несколько языковых расширений, поддерживающих синтаксис атрибутов в коде: GNU `__attribute__((...))` [3] и MSVC `__declspec(...)` [4]. Помимо расширений, современные стандарты языков C и C++ описывают атрибуты `[[...]]`, которые можно добавлять практически к любой сущности в коде.

С помощью атрибутов программист может влиять на различные аспекты работы компилятора: на вывод диагностики, например с помощью атрибутов `deprecated` и `fallthrough`, или непосредственно на генерацию машинного кода (с помощью атрибута `omp::parallel`). Таким образом, в зависимости от типа атрибута, компилятор может как сохранять информацию о нем в используемом им промежуточном представлении, так и не сохранять ее. Например, GNU-атрибут `nonnull` в компиляторе Clang непосредственно сохраняется как `nonnull`-атрибут параметра функции в LLVM IR [5].

2.2 JVM-аннотации

Аннотации в языках на основе JVM (Java Virtual Machine), в том числе в Java и Kotlin, позволяют предоставлять метаданные для сущностей в коде. Они применяются к использованиям типов, а также к определениям (в частности, к определениям типов, методов, полей, переменных и параметров) [6].

Аннотации могут присутствовать в исходном коде и удаляться после этапа компиляции – такие аннотации могут обрабатываться самим компилятором, его плагинами [7] или процессорами аннотаций [8] для обнаружения ошибок или генерации дополнительного кода. Также аннотации могут сохраняться в JVM-байткоде и быть доступны во время выполнения программы при помощи инструментов Reflection API [9], например, для средств метапрограммирования.

В языке Kotlin, в отличие от Java, аннотации можно применять и к произвольным выражениям. Но такие аннотации доступны только в момент компиляции, в байткоде они не сохраняются. Единственный способ их обработать – плагины компилятора или процессоры аннотаций [10-11].

Система типов языка Kotlin имеет разделение типов на те, значениями которых может являться `null` (nullable типы) и те, значениями которых `null` быть не может (non-nullable типы) [12]. Однако байткод JVM такого разделения не имеет – все переменные ссылочных типов могут иметь значения `null`, поэтому для предоставления данной информации о типах другим программам, компилятор языка Kotlin сохраняет её в байткоде в виде JVM-аннотаций типов.

3. Использование аннотаций в статическом анализаторе

Аннотирование исходного кода может использоваться с разными целями. Мы рассмотрели следующие случаи:

1. Выступать источником ошибки в том случае, если реальный источник неизвестен (или труден для получения) статическому анализатору.
2. Предоставлять дополнительную семантику о программе, которая может в явном виде не присутствовать в исходном коде.
3. Быть некоторыми инвариантами программы, которые требуется дополнительно проверить статическому анализатору.

Далее подробно опишем каждый из сценариев использования.

3.1 Аннотации как источник ошибки

Популярным подходом к статическому анализу является анализ на основе резюме, при котором выполняется обход функций программы по графу вызовов снизу-вверх начиная с листьев графа. При этом анализатору доступна информация о вызываемых функциях, но ничего не известно о вызывающих.

Помимо этого, функции могут быть публичной частью какой-либо библиотеки и не вызываться нигде в исходном коде. В таком случае информации о контексте вызова вовсе нет.

Пользовательские аннотации позволяют предоставить дополнительную информацию о входных параметрах функций, фактически о требованиях, предъявляемых к контексту вызова функции. Аннотация `Nullable` из листинга 1 является примером такой аннотации. В данном случае сообщается, что в контексте вызова допустимо передавать нулевую ссылку. Поэтому статический анализатор может считать, что функция вызывается с передачей значения `null` в качестве аргумента. Если функция разыменовывает параметр без проверки, то будет выдано сообщение об ошибке.

```
1  public class Example {
2    private final Map<Long, String> idToUserName;
3
4    public void store(long id, @Nullable User user) {
5      idToUserName.put(
6        id,
7        user.getName() // Потенциальное разыменование нулевой ссылки
8      );
9    }
10 }
```

Листинг 1. Пример аннотации Nullable в Java.
Listing 1. Example of Nullable annotation for Java.

3.2 Предоставление дополнительной семантики о программе

Значения полей структур или классов могут быть связаны друг с другом, при этом не всегда такие зависимости могут быть выражены средствами языка программирования.

В листинге 2 показан пример структуры, имеющей два поля: `len` и `buf`. В поле `len` хранится размер массива, на который указывает поле `buf`. Функция `test`, заполняющая массив, содержит ошибку: цикл выходит за границы массива. Проверку `i <= s->len` необходимо заменить на `i < s->len`.

```
1  typedef struct {
2      unsigned int len;
3      char* buf __attribute__((buflen(len)));
4  } S;
5
6  void test(S* s) {
7      for (int i = 0; i <= s->len; ++i) {
8          s->buf[i] = i; // ERROR: Выход за границы при i = len
9      }
10 }
```

Листинг 2. Массив и его длина в качестве полей структуры.
Listing 2. Array and its length as structure fields.

В некоторых случаях статический анализатор может отследить заполнение полей структур и выдать ошибку в месте использования. Но код заполнения может находиться в программе далеко от места его использования, и статическому анализатору будет сложно отследить путь в программе между этими точками, а также проверить выполнимость пути. Наличие атрибута `buflen(len)` позволило бы найти эту ошибку.

В общем случае размер массива может определяться более сложным способом. Размер может быть на единицу больше, чем `len`, если `len` не учитывает завершающий ноль строки. Либо

размер может быть произведением длины массива и размера отдельного элемента. В листинге 3 показан пример более сложной структуры, в которой размер массива определяется по формуле `header + height * width`.

```
typedef struct {
    unsigned int header;
    unsigned int height;
    unsigned int width;
    char* buf __attribute__((buflen(header + height * width)));
} S;

void fill(S* s, unsigned int row, unsigned int col, char val) {
    unsigned int index = s->header + row * s->width + col;
    if (index <= s->header + s->height * s->width) {
        s->buf[index] = val; // ERROR: Выход за границы буфера
    }
}
```

Листинг 3. Формула над полями.
Listing 3. Formula over fields.

Другим примером может служить аннотация `GuardedBy` из Android API [13]. Данная аннотация позволяет указывать, что обращение к члену класса должно происходить только при синхронизации на поле с соответствующим именем. Статический анализатор может выдать ошибку, если обращение к полю производится без синхронизации.

```
public class Example {
    private final Object lock = new Object();

    @GuardedBy("lock")
    private boolean field;

    public void test_ok() {
        synchronized (lock) {
            field = false; // OK
        }
    }

    public void test_bad() {
        field = true; // ERROR: Отсутствует синхронизация
    }
}
```

Листинг 4. Аннотация `GuardedBy`.
Listing 4. `GuardedBy` annotation.

3.3 Проверка корректности инвариантов

Также полезным будет реализация проверок, что переменным, помеченным аннотациями, присваиваются допустимые значения. В этом случае пользователь программы получит предупреждение от анализатора, если присваивающий код нарушает инварианты, заданные аннотациями. В листинге 5 показан пример кода, где поле, аннотированное

интервалом `[-10; 10]`, получает значение за пределами этого интервала. Выдача предупреждения позволит обнаружить ошибку в коде.

```
1  typedef struct {
2      int value __attribute__((value_interval(-10, 10)));
3  } S;
4
5  void test(S* s) {
6      s->value = 200; // ERROR: Выход за границы интервала
7  }
```

Листинг 5. Нарушение инварианта структуры.
Listing 5. Structure invariant violation.

4. Реализация

4.1 Clang

Первым этапом мы определили, какую дополнительную информацию было бы неплохо иметь в анализаторе, и составили соответствующий список атрибутов, приведённый в таблице 1 (разумеется, позже могут быть добавлены и другие атрибуты).

Табл. 1. Атрибуты C/C++.
Table 1. C/C++ attributes.

Название атрибута	Семантика
`buflen(x)`	поле структуры хранит буфер, размер которого задаётся выражением `x`
`value_interval(l, r)`	число принадлежит интервалу `[l; r]`
`taint`	адрес памяти может контролироваться злоумышленником
`taint_int/taint_int(l, r)`	целое число в интервале `[l; r]` может контролироваться злоумышленником
`not_null`	указатель не может быть нулевым
`possible_null`	указатель может быть нулевым
`possible_negative`	целое число может быть отрицательным
`target_dependent`	значение зависит от используемой платформы (например, размер типа `int`)

Можно заметить, что `not_null` очень похож на GNU-атрибут `nonnull`, но новый атрибут применим не только к функциям, но и к переменным и полям структур и классов.

Компилятор Clang, используемый в Svace, уже модифицирован для того, чтобы распознавать различные языковые расширения, поддерживаемые другими компиляторами. Таким образом, мы модифицировали Clang и для того, чтобы он распознавал наши новые атрибуты.

Большинство атрибутов довольно просты в реализации, так как имеют аргументы-константы или не имеют аргументов совсем. Тем не менее, даже для простых атрибутов все же были сделаны проверки правильности использования: корректность интервалов, правильность типов сущностей, к которым указаны атрибуты, а также отсутствие противоречивых атрибутов (`possible_null` и `not_null` не могут быть использованы для одной сущности).

Наибольшую сложность для реализации по нескольким причинам представлял атрибут buflen. Первой причиной было то, что в аргументе атрибута должно быть возможно использовать поля, которые будут объявлены в теле класса позднее (как в листинге 6). То есть атрибут должен обрабатываться после того, как компилятор проведёт синтаксический разбор класса до конца. Как известно, так же должны обрабатываться C++ inline-методы, определенные в теле класса, однако в нашем случае атрибут может быть использован и в C-коде, для которого Clang не поддерживал отложенный разбор. Мы распространили C++ механизм отложенного разбора на язык C, модифицировав функции синтаксического разбора структур так, чтобы во время анализа аргументов `buflen` было доступно полное определение класса.

```
1 struct B {
2     char* buf [[buflen(len)]];
3     size_t len;
4 };
```

Листинг 6. Отложенный синтаксический разбор.
Listing 6. Delayed parsing.

Вторая причина также происходила из правил языка C. В отличие от C++, C не поддерживает использование полей структур без явного указания объекта, к которому относятся поля [14], но в контексте атрибута не существует никакого объекта, который можно использовать. Таким образом, было необходимо реализовать поддержку генерации абстрактного синтаксического дерева в компиляторе Clang для прямого использования полей в аргументах атрибута, что и было проделано.

Третья причина следовала из особенностей работы анализатора Svace. Svace разработан так, чтобы перехватывать обычную сборку проекта [1], следовательно новые атрибуты не должны нарушать работу оригинальной сборки. В случае со стандартными атрибутами (`[[...]]`) никаких проблем нет, так как стандарт говорит, что неизвестные атрибуты должны игнорироваться, а аргументом атрибута может быть произвольная последовательность токенов [14]. Для GNU-атрибутов (`__attribute__`) правила другие – неизвестные атрибуты по-прежнему игнорируются, однако аргументы неизвестных атрибутов должны иметь одну из допустимых форм: идентификатор; идентификатор, за которым следует несколько выражений, разделенных запятой; несколько или ноль выражений, разделенных запятой [3]. Аргументы `buflen` могут быть сложными выражениями (как было показано в листинге 3) и могут вызывать ошибки в оригинальной сборке, которая обрабатывает `buflen` как неизвестный атрибут. Для избежания таких ошибок аргумент `buflen` может быть экранирован двойными кавычками (см. листинг 7), чтобы оригинальный компилятор обрабатывал его как строковую константу (которая, конечно, является и выражением).

```
1 struct B {
2     char* buf __attribute__((buflen("2*len*len+1")));
3     size_t len;
4 };
```

Листинг 7. Экранированный аргумент.
Listing 7. Escaped argument.

4.2 JVM

Анализ JVM-программ в Svace реализован на основе байткода JVM, генерируемого при компиляции [15-17].

С одной стороны, поддержка пользовательских аннотаций при таком подходе довольно проста – для аннотаций должна быть указана опция (Retention), при которой информация об их использовании сохраняется в байткод. Модификаций в компилятор при этом вносить не требуется, в отличии от случая с атрибутами для C/C++.

С другой стороны, невозможно получить информацию об аннотациях, которые существуют только в момент компиляции, но не добавляются в байткод. В частности, как сказано в разделе 2.2, аннотации на произвольных выражениях из Kotlin сохранить в байткод невозможно. Тем не менее, для нашего подхода мы считаем такое ограничение несущественным – интересные для анализа в Svace свойства имеет смысл указывать при помощи аннотаций только для локальных переменных, параметров функций, их возвращаемых значений, а также для полей классов.

Для JVM были поддержаны аннотации, приведённые в табл. 2.

Табл. 2. Аннотации JVM.
Table 2. JVM annotations.

Название аннотации	Семантика
`Interval(l, r)`	число принадлежит интервалу `[l; r]`
`TaintedPtr`	ссылочное значение может контролироваться злоумышленником
`TaintedInterval(l, r)`	целое число в интервале `[l; r]` может контролироваться злоумышленником
`Trusted`	значение используется в чувствительной операции
`TrustedInterval(l, r)`	целочисленное значение используется в чувствительной операции при принадлежности интервалу `[l; r]`
`NotNull`	ссылочное значение не может принимать значение `null`
`Nullable`	ссылочное значение может принимать значение `null`

Как было сказано в разделе 2.2, информация о типах из Kotlin, допускающих значение `null`, тоже сохраняется в байткод JVM. Поэтому их поддержка эквивалентна поддержке аннотаций `NotNull` и `Nullable`.

4.3 Анализатор

В Svace анализ основан на отслеживании интересующих свойств на символьных переменных. Реализация детекторов заключается в проверке свойств для соответствующих инструкций (например, проверка свойства указателя быть нулевым для инструкции разыменования). Реализация большинства атрибутов является тривиальной. Для входных параметров функции она заключается в установке нужных свойств в начале анализа каждой функции. Для полей структур свойства устанавливаются при чтении соответствующего поля.

Нарушение инвариантов проверяется при присваивании полям структур и при передаче аргументов в функции некорректных (на основе соответствующих анализируемых свойств) значений.

Отдельный интерес представляет реализация `buflen`. При генерации LLVM IR для структуры, если её поле помечено атрибутом `buflen`, создается функция `struct_name.buf_name()`, которая в качестве аргумента принимает указатель на структуру `struct_name`, а возвращает результат выражения, указанного в качестве аргумента атрибута `buflen`. Идентификатор этой функции добавляется в отладочную информацию для поля `buf_name`. После чтения отладочной информации анализатор сопоставляет указатель на

функцию `struct_name.buf_name()` полю структуры. В листинге 8 показан пример такой кодогенерации, соответствующий исходному коду из листинга 3.

```
1  typedef struct {
2      unsigned int header;
3      unsigned int height;
4      unsigned int width;
5      char* buf __attribute__((buflen(S.buf)));
6  } S;
7
8  unsigned int S.buf(S* s) { // Сгенерировано компилятором
9      return s->header + s->height * s->width;
10 }
11
12 void fill(S* s, unsigned int row, unsigned int col, char val) {
13     unsigned int index = s->header + row * s->width + col;
14     if (index <= s->header + s->height * s->width) {
15         s->buf[index] = val;
16     }
17 }
```

Листинг 8. Формула над полями с созданной функцией.
Listing 8. Formula over fields with created function.

Идея заключается в следующем. Изначально выполняется анализ всех служебных функций типа `S.buf`. Проведенный анализ позволяет установить, что возвращаемое значение является результатом арифметического выражения, зависящего от полей структуры входного параметра: `s->header + s->height * s->width`. Далее анализ выполняется на основе резюме стандартным образом, но при анализе функции `fill` в её начало добавляется вызов функции `S.buf`. Его обработка заключается в применении резюме этой функции и трансляции символьного выражения для возвращаемого значения в контекст `fill`. Результирующее символьное выражение сопоставляется с полем структуры, помеченным атрибутом `buflen`, то есть с `s->buf`.

При обращении к данному полю структуры по индексу (как на строке 15 в примере) или при обращении к полю с помощью функций `memcpy` и `memcmp` проверяется возможность выхода за указанную длину буфера в данной точке программы. Для этого составляется формула ошибки для SMT-решателя:

$$max(index) \geq len \wedge len \geq 0$$

где $\square\square\square$ – идентификатор значения из анализируемого свойства, а $\square\square\square(\square\square\square\square\square)$ – максимальное значение индекса из интервала допустимых значений, по которому обращаются к буферу (или число элементов для копирования или сравнения в случае с `memcpy` и `memcmp` соответственно). Если формула выполнима, то выдаётся соответствующее предупреждение о возможности выхода за границы буфера.

5. Результаты

Помимо аннотаций, добавленных специально как часть Svace, анализатор также учитывает все аннотации с названием `NotNull`, `NonNull` и `Nullable` из любой библиотеки как специальные. Мы проанализировали часть исходного кода операционной системы Android-14, написанной на Java, и исходный код компилятора Kotlinc версии 1.9.22, написанного на

Java и Kotlin, в которых повсеместно используются подобные аннотации (в частности, и неявно для типов из Kotlin, допускающих значение `null`). В таблице 3 приведены результаты тестирования этих аннотаций.

До изменений на Android-14 выдавалось суммарно 276822 предупреждения. После поддержки перечисленных аннотаций появилось 2904 новых срабатывания и пропало 21 срабатывание. Из пропавших срабатываний 8 было истинными и 13 было ложными. Из 100 случайных новых срабатываний 13 являются ложными, 54 – истинными и 33 – не попадают ни в одну категорию.

До изменений на Kotlinc-1.9.22 выдавалось суммарно 3682 предупреждения. После поддержки перечисленных аннотаций появилось 2879 новых срабатываний и пропало 94 срабатывания. Из пропавших срабатываний 12 было истинными и 82 ложными. Из 100 случайных новых срабатываний 15 являются ложными, 25 – истинными и 60 – не попадают ни в одну категорию.

Табл. 3. Результаты для аннотаций, связанных с null, на JVM-проектах.
Table 3. Results for null-related annotations in JVM projects.

Проект	Всего предупреждений		Появилось			Пропало		
	До	После	Всего	Истин. (из 100)	Ложн. (из 100)	Всего	Истин.	Ложн.
Android-14	276822	279705	2904	54	13	21	8	13
Kotlinc-1.9.22	3682	6467	2879	25	15	94	12	82

33 новых срабатывания на Android-14 и 60 новых срабатываний на Kotlinc-1.9.22, которые не попадают в категории истинных или ложных, являются истинными формально, но фактически разыменования нулевой ссылки в таком коде никогда не происходит. Причина этому в том, что не всегда аннотация `@Nullable` может являться источником ошибки. Например, в листинге 9 приведён пример, когда метод, помеченный данной аннотацией, возвращает нулевую ссылку только при определённом условии. Если это условие предварительно проверяется, дефект с разыменованием нулевой ссылки не проявляется. Использование аннотации `@Nullable` в подобных контекстах довольно распространено, что может приводить к большому количеству срабатываний, которые с точки зрения анализатора являются корректными, т.к. явная проверка на `null` отсутствует, но не релевантны для пользователя. Для избежание таких случаев полезным было бы введение аннотации, подобной `@Nullable`, но хранящей и условие, при котором возможно нулевое значение, как, например, в стандартной библиотеке языка C# [18].

6. Похожие работы

Среда разработки IntelliJ IDEA [19] предоставляет возможности для статического анализа кода с учётом наиболее распространённых аннотаций. Помимо этого, для IntelliJ IDEA предоставляются отдельные специальные аннотации [20]. Наиболее интересной из них является аннотация `@Contract`, позволяющая задавать контракт, которому следует и аннотированный метод, и все его вызовы: например, что функция является чистой; или же что функция бросает исключение при выполнении условия на входные параметры. Пример использования приведён в листинге 10.

```
1 interface List<T> {
2     boolean isEmpty();
3
4     @Nullable // Возвращает null только для пустого списка
```

```
    T getFirstOrNull();
}

class Example {
    public void test(List<User> users) {
        if (users.isEmpty()) return;
        User firstUser = users.getFirstOrNull();
        String name = firstUser.getName(); // Нет ошибки
        ...
    }
}
```

Листинг 9. Отсутствие разыменования нулевой ссылки при наличии `@Nullable`.
Listing 9. Absent null dereference issue in presence of `@Nullable`.

```
public class Example {
  // Чистая функция, результат которой обязан использоваться
  @Contract(pure = true)
  public static boolean not(boolean value) {
    return !value;
  }

  // Функция, которая завершается исключением,
  //   если значение входного параметра ложно
  @Contract("false -> fail")
  public static require(boolean condition) {
    if (!condition) throw new IllegalStateException();
  }
}
```

Листинг 10. Пример использования аннотации `Contract`.
Listing 10. Example of `Contract` annotation usage.

Язык Python начиная с версии 3.5 предоставляет возможность указания типовых аннотаций [21-22]. Данные аннотации не являются обязательными, но могут быть использованы средами разработки, линтерами и другими инструментами для статической проверки корректности типов. Они стали основой для статического анализатора Муру [23]. В листинге 11 на строке 1 `int` – это не тип переменной `b`, а аннотация, которая сообщает, что переменная `b` должна иметь тип `int`. На строке 5 переменная получает строковый тип. С точки зрения языка это корректный код, и он будет успешно исполнен. Но анализатор Муру выдаст ошибку на этой строке, так как переменная получает тип, который противоречит аннотации. Подобные аннотации позволяют добавить проверку типов для языка с динамической типизацией и обнаружить часть ошибок во время написания программы. Согласно исследованию [24], 7% проектов на Python используют аннотации типов. На наш взгляд это довольно высокий показатель, так как язык Python занимает нишу для быстрого создания прототипов, и часто его используют из-за отсутствия строгой типизации.

```
b: int
if f:
    b = int(input())
else:
```

```
5     b = input()
```

Листинг 11. Пример типовых аннотаций в Python.
Listing 11. Python type annotations example.

Clang Static Analyzer [25] позволяет аннотировать исходный код при помощи атрибутов GCC [3], таким образом помогая анализатору находить новые дефекты и уменьшать количество ложных срабатываний. Пример приведён в листинге 12. Атрибут `nonnull(1,3)` показывает, что первый и третий параметр никогда не должны принимать нулевое значение. Но на строке 4 в качестве последнего аргумента передаётся нулевой указатель, о чём сообщит анализатор.

```
1 int bar(int*p, int q, int *r) __attribute__((nonnull(1,3)));
2
3 int foo(int *p, int *q) {
4    return !p ? bar(q, 2, p)
5              : bar(p, 2, q);
6 }
```

Листинг 12. Пример аннотаций в CSA.
Listing 12. CSA source annotations example.

В инструменте Svace реализована реакция на комментарии в коде. Плюсы – универсальность реализации, подходит для множества языков. Минусы – реализация основана на синтаксическом разборе исходного кода, компилятор не используется, в сложных случаях возможны ошибки. Пример приведён в листинге 13.

```
1 typedef struct {
2   int* ptr; // svace: possible_null
3 } S;
4
5 int example(S* s) {
6   return *s->ptr; // Потенциальное разыменование нулевого
7 указателя
  }
```

Листинг 13. Пример комментария-аннотации.
Listing 13. Example of comment annotation.

Другой пример аннотаций в виде комментариев реализован в легковесном статическом анализаторе Splint [26], в котором используются аннотации для параметров функций, возвращаемых значений, глобальных переменных и полей структур в виде комментариев к исходному коду. Например, аннотация для параметра `/*@notnull@*/` означает, что он не может принимать нулевое значение. Инструмент выдаст предупреждения во всех случаях, когда параметр может получить нулевое значение. При анализе таких функций анализатор считает, что значение параметра ненулевое. При этом аннотировать можно не только определения функций, но и их объявления, что позволяет аннотировать библиотечные функции с отсутствующим кодом. В Svace мы не вводили такую возможность, т.к. в инструменте уже присутствует механизм, позволяющий моделировать поведение библиотечных функций.

Работа [27] посвящена поиску уязвимостей форматной строки в C с помощью вывода типов. Помеченные данные реализованы в виде расширения системы типов C дополнительными квалификаторами типов. Квалификатор tainted используется подобно уже существующему квалификатору const. Предупреждение выдаётся, если выражение с таким квалификатором

используется как форматная строка. Предложенная система похожа на атрибуты для C/C++ в Svace, отличия заключаются в том, что они используются разными видами анализа, а также тем, что дополнительные квалификаторы вызовут ошибки компиляции и поэтому требуют специальной подготовки кода.

7. Заключение

Статический анализатор, способный проводить проверки без специальной подготовки программы, обладает очевидными преимуществами. Тем не менее, возможность добавления пользователем дополнительной информации о программе также представляет ценность. Ключевым достоинством аннотаций является автоматическая проверка анализатором указанных свойств при их нарушении, что не только упрощает анализ программы, но и повышает читаемость кода.

В данной работе рассмотрены возможности аннотирования исходного кода для языков C, C++, Java и Kotlin, а также особенности их реализации в статическом анализаторе. Пользовательские аннотации позволяют существенно улучшить процесс анализа, однако их эффективность во многом зависит от корректности и полноты предоставляемых пользователем данных.

Список литературы / References

[1]. Иванников В. П., Белеванцев А. А., Бородин А. Е., Игнатьев В. Н., Журихин Д. М., Аветисян А. И., Леонов М. И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7.

[2]. Бородин А. Е., Белеванцев А. А. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 111-134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[3]. GNU Compiler Collection. Attribute Syntax. Английский. URL: https://gcc.gnu.org/onlinedocs/gcc/Attribute-Syntax.html (дата обращения 16.04.2025).

[4]. __declspec | Microsoft Learn. Английский. URL: https://learn.microsoft.com/en-us/cpp/cpp/declspec?view=msvc-170 (дата обращения 16.04.2025).

[5]. LLVM Compiler Infrastructure. Английский. URL: https://llvm.org/docs/LangRef.html (дата обращения 16.04.2025).

[6]. Gosling J., Joy B., Steele G., Bracha G., Buckley A., Smith D., Bierman G. The Java® Language Specification. Java SE 21 Edition. 2023 г.

[7]. Java SE 21. Interface Plugin. Английский. URL: https://docs.oracle.com/en/java/javase/21/docs/api/jdk.compiler/com/sun/source/util/Plugin.html (дата обращения 15.04.2025).

[8]. Java SE 21. Interface Processor. Английский. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.compiler/javax/annotation/processing/Processor.html (дата обращения 15.04.2025).

[9]. The Java™ Tutorials. Trail: The Reflection API. Английский. URL: https://docs.oracle.com/javase/tutorial/reflect/ (дата обращения 15.04.2025).

[10]. Kotlin Symbol Processing API. Английский. URL: https://kotlinlang.org/docs/ksp-overview.html (дата обращения 11.04.2025).

[11]. kapt compiler plugin. Английский. URL: https://kotlinlang.org/docs/kapt.html (дата обращения 11.04.2025).

[12]. Kotlin. Null safety. Английский. URL: https://kotlinlang.org/docs/null-safety.html (дата обращения 24.04.2025).

[13]. Android API Reference. GuardedBy. Английский. URL: https://developer.android.com/reference/androidx/annotation/GuardedBy (дата обращения 22.04.2025).

[14]. International Standard ISO/IEC 14882:2024(E) – Programming Language C++. 2024 г.

[15]. Меркулов А. П., Поляков С. А., Белеванцев А. А. Анализ программ на языке Java в инструменте Svace. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 57-74. DOI: 10.15514/ISPRAS-2017-29(3)-5.

[16]. Афанасьев В. О., Поляков С. А., Бородин А. Е., Белеванцев А. А. Kotlin с точки зрения разработчика статического анализатора. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр.67-82. DOI: 10.15514/ISPRAS-2021-33(6)-5.
[17]. Афанасьев В. О., Бородин А. Е., Белеванцев А. А. Статический анализ для языка Scala. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 9-20. DOI: 10.15514/ISPRAS-2024-36(3)-1.
[18]. Attributes for null-state static analysis interpreted by the C# compiler. Conditional post-conditions: NotNullWhen, MaybeNullWhen, and NotNullIfNotNull. Английский. URL: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/nullable-analysis#conditional-post-conditions-notnullwhen-maybenullwhen-and-notnullifnotnull (дата обращения 14.05.2025).
[19]. IntelliJ IDEA. Annotations. Английский. URL: https://www.jetbrains.com/help/idea/annotating-source-code.html (дата обращения 15.04.2025).
[20]. IntelliJ IDEA. Annotations. JetBrains annotations dependency. Английский. URL: https://www.jetbrains.com/help/idea/annotating-source-code.html#jetbrains-annotations (дата обращения 15.04.2025).
[21]. The Python Standard Library. typing – Support for type hints. Английский. URL: https://docs.python.org/3/library/typing.html (дата обращения 15.04.2025).
[22]. Van Rossum G., Lehtosalo J., Langa L. Python Enhancement Proposals. PEP 484 – Type Hints. Английский. URL: https://peps.python.org/pep-0484/ (дата обращения 15.04.2025).
[23]. Mypy: Optional Static Typing for Python. Английский. URL: https://mypy-lang.org/ (дата обращения 15.04.2025).
[24]. Di Grazia L., Pradel M. The Evolution of Type Annotations in Python: An Empirical Study. Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2022 г. DOI: 10.1145/3540250.3549114.
[25]. Clang 21.0.0git documentation. Source Annotations. Английский. URL: https://clang.llvm.org/docs/analyzer/user-docs/Annotations.html (дата обращения 16.04.2025).
[26]. Evans D., Larochelle D. Improving security using extensible lightweight static analysis. IEEE software, том 19, вып. 1, 2002 г., стр. 42-51. DOI: 10.1109/52.976940.
[27]. Shankar U., Talwar K., Foster J. S., Wagner D. Detecting format string vulnerabilities with type qualifiers. 10th USENIX Security Symposium (USENIX Security 01), 2001 г.

Информация об авторах / Information about authors

Виталий Олегович АФАНАСЬЕВ – аспирант ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, JVM языки.

Vitaly Olegovich AFANASYEV – postgraduate student at ISP RAS. Research interests: compiler technologies, static analysis, JVM languages.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), researcher at ISP RAS. Research interests: static analysis for finding errors in source code.

Евгений Александрович ВЕЛЕСЕВИЧ – научный сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии.

Evgeny Alexandrovich VELESEVICH – researcher at ISP RAS. Research interests: compiler technologies.

Борис Викторович ОРЛОВ – студент ВМК МГУ, сотрудник ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ.

Boris Viktorovich ORLOV – student at the Faculty of Computational Mathematics and Cybernetics of Lomonosov Moscow State University, researcher at ISP RAS. Research interests: compiler technologies, static analysis.

DOI: 10.15514/ISPRAS-2025-37(6)-9

Распознавание заголовков таблиц на основе больших языковых моделей

И.И. Охотин, ORCID: 0009-0003-9600-913X <ilia.ohotin@yandex.ru>
Н.О. Дородных, ORCID: 0000-0001-7794-4462 <nikidorny@icc.ru>
Институт динамики систем и теории управления имени В.М. Матросова СО РАН, Россия, 664033, г. Иркутск, ул. Лермонтова, д. 134

Аннотация. Автоматическое распознавание заголовков таблиц остается сложной задачей из-за разнообразия макетов, включая многоуровневые заголовки, объединенные ячейки и нестандартное форматирование. В данной статье впервые предложена методология оценки эффективности больших языковых моделей в решении этой задачи с использованием текстовых подсказок (промпт-инжиниринга). Исследование охватывает восемь различных моделей и шесть стратегий текстовых подсказок, от минималистичных (zero-shot) до сложных с примерами (few-shot), на выборке из 237 таблиц. Результаты демонстрируют, что размер модели критически влияет на точность: крупные модели (405 млрд. параметров) достигают F-меры ≈ 0.80–0.85, тогда как малые (7 млрд. параметров) показывают F1 ≈ 0.06–0.30. Усложнение текстовых подсказок за счет пошаговых инструкций, критериев поиска и примеров улучшает результаты только для крупных моделей, тогда как для малых приводит к деградации из-за перегруженности контекста. Наибольшие ошибки возникают при обработке таблиц с иерархическими заголовками и объединенными ячейками, где даже средние и крупные модели теряют точность распознавания. Практическая значимость работы заключается в выявлении оптимальных конфигураций текстовых подсказок для разных типов моделей. Например, для крупных моделей эффективны краткие инструкции, а для средних – пошаговые инструкции с критериями поиска. Данное исследование открывает новые возможности по созданию универсальных инструментов для автоматического анализа заголовков таблиц.

Ключевые слова: таблица; заголовки таблицы; распознавание структуры таблиц; распознавание заголовков; большая языковая модель; текстовые подсказки.

Для цитирования: Охотин И.И., Дородных Н.О. Распознавание заголовков таблиц на основе больших языковых моделей. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 149–166. DOI: 10.15514/ISPRAS–2025–37(6)–9.

Благодарности: Работа выполнена в рамках государственного задания Министерства науки и высшего образования Российской Федерации (тема № 1023110300006-9).

Using Large Language Models for Table Header Recognition

I.I. Okhotin, ORCID: 0009-0003-9600-913X <ilia.ohotin@yandex.ru>
N.O. Dorodnykh, ORCID: 0000-0001-7794-4462 <nikidorny@icc.ru>

Matrosov Institute for System Dynamics and Control Theory
of the Siberian Branch of Russian Academy of Sciences (ISDCT SB RAS),
134, Lermontov st., Irkutsk, 664033, Russia.

Abstract. Automatic table header recognition remains a challenging task due to the diversity of table layouts, including multi-level headers, merged cells, and non-standard formatting. This paper is the first to propose a methodology to evaluate the performance of large language models on this task using prompt engineering. The study covers eight different models and six prompt strategies with zero-shot and few-shot settings, on a dataset of 237 tables. The results demonstrate that model size critically affects the accuracy: large models (405 billion parameters) achieve F1 ≈ 0.80–0.85, while small ones (7 billion parameters) show F1 ≈ 0.06–0.30. Complicating prompts with step-by-step instructions, search criteria, and examples improves the results only for large models, while for small ones it leads to degradation due to context overload. The largest errors occur when processing tables with hierarchical headers and merged cells, where even large models lose up to accuracy of recognition. The practical significance of this paper lies in identifying optimal configurations of prompts for different types of models. For example, short instructions are effective for large models, and step-by-step instructions with search criteria are effective for medium ones. This study opens up new possibilities for creating universal tools for automatic analysis of table headers.

Keywords: table; table headers; table structure recognition; header recognition; large language model; prompt engineering.

For citation: Okhotin I.I., Dorodnykh N.O. Using large language models for table header recognition. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 149-166 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-9.

Acknowledgements. This work was supported by the state assignment of Ministry of Science and Higher Education of the Russian Federation (theme No. 1023110300006-9).

1. Введение

Таблицы служат универсальным инструментом для структурированного представления данных в научных публикациях, финансовых отчетах, веб-документах и других областях. Их способность компактно отображать сложные взаимосвязи между элементами делает их незаменимыми для анализа и принятия решений. Однако автоматическая обработка таблиц остается сложной задачей из-за разнообразия их макетов: объединенные ячейки, многоуровневые заголовки, наличие пустых областей, нестандартное форматирование и отсутствие единых правил оформления. Эти особенности затрудняют как физическое распознавание структуры таблиц (например, границ строк и столбцов), так и семантическую интерпретацию заголовков, которые часто определяют контекст данных. При этом автоматическое распознавание заголовков таблиц актуально для таких приложений, как извлечение данных, построение схем баз данных и предварительная обработка информации для машинного обучения.

Современные подходы к распознаванию структуры таблиц, включая методы на основе свёрточных и графовых нейронных сетей, а также предварительно обученные языковые модели, основанные на архитектуре трансформер, демонстрируют прогресс в этой области [1-2], но сталкиваются с ограничениями при работе с иерархическими заголовками и шумными данными. Появление больших языковых моделей открыло новые возможности для обработки табличных данных за счёт своей способности более глубоко анализировать контекст и семантику данных [3]. Однако их эффективность зависит от стратегий взаимодействия с использованием текстовых подсказок (так называемого, промпт-

инжиниринга – *prompt engineering*), который до сих пор недостаточно изучен в контексте распознавания заголовков таблиц.

В данной статье предлагается новая методология автоматического распознавания заголовков таблиц, обладающих разной структурной компоновкой, на основе использования различных больших языковых моделей. В частности, основной вклад данной работы состоит:

- В оценке влияния различных видов текстовых подсказок, размеров больших языковых моделей и структурной сложности таблиц на точность распознавания заголовков. Так в исследовании протестированы 8 моделей (от 7 млрд. до 405 млрд. параметров) с шестью типами текстовых подсказок, включая *zero-shot* и *few-shot* подходы.
- Впервые получены экспериментальные оценки производительности больших языковых моделей в контексте решения задачи распознавания заголовков таблиц. Эксперименты проведены на выборке из 237 таблиц. Результаты показывают, что даже средние и крупные модели требуют адаптивных стратегий формирования текстовых подсказок для обработки нестандартных макетов таблиц, а их эффективность напрямую зависит от минимализма и ясности формулировок текстовых подсказок.

Статья организована следующим образом: Раздел 2 представляет современное состояние исследований в области автоматического распознавания структуры таблиц. В разделе 3 описывается методология исследования, включая формальную постановку задачи, подготовку данных и другие детали. Раздел 4 включает описание экспериментов, их результаты и обсуждение. В заключении делается краткий обзор полученных результатов, а также приводятся планы будущей работы.

2. Современное состояние исследований

Таблицы являются распространённым и достаточно эффективным средством для представления и хранения структурированных данных во многих предметных областях. Однако они в первую очередь предназначены для интерпретации человеком, что делает их автоматическую обработку проблематичной. Область, занимающаяся автоматическим пониманием табличной информации (*table understanding*) [4-5], рассматривает следующие основные проблемы обработки таблиц:

1) Обнаружение (поиск и идентификация) таблиц в исходном информационном источнике, например, электронном документе или изображении (например, скан документа).
2) Распознавание физической структуры таблиц посредствам идентификации строк, столбцов и ячеек (восстановление матричной сетки) в обнаруженной таблице, которая обычно рассматривается как входное изображение.
3) Восстановление логической структуры таблиц с целью идентификации метаданных (заголовков), которые описывают некоторое подмножество значений ячеек данных в таблице.
4) Семантическое аннотирование содержимого таблиц понятиями из внешнего словаря (модели предметной области, онтологии, или графа знаний) для восстановления отсутствующей семантики данных.

Проблемы распознавания физической и логической структуры таблиц (*table structure recognition*) в известной научной литературе обычно рассматриваются вместе. Существует два основных направления к разработке методов, решающих эти проблемы:

1) Методы, основанные на правилах, например, [6-8], опираются на правила анализа и интерпретации таблиц. Такие решения обычно не охватывают все многообразие

макетов таблиц, форматирования и содержимого. Они, как правило, ограничены обычными макетами, атомарными ячейками и плоскими заголовками, игнорируя случаи, когда эти предположения не выполняются, как обсуждалось в [5, 9].

2) Методы, управляемые данными. Такие решения могут использовать как традиционные модели на основе машинного обучения, например, метод опорных векторов (*Support Vector Machine, SVM*) и случайны лес (*Random Forest*) [10], или кластерный анализ [11], так и модели на основе глубокого обучения, такие как [12-25].

Ранние методы распознавания, основанные на правилах и анализе макета с опорой на техники OCR (*Optical Character Recognition*), показывали невысокую степень адаптивности к разнообразию стилей таблиц и шуму распознавания текста. С приходом глубокого обучения появились решения на основе архитектуры свёрточных нейронных сетей (*Convolutional Neural Network, CNN*), например, модель TableNet [12], которая в режиме «*end-to-end*» одновременно детектирует область таблицы и сегментирует строки и столбцы, достигая передовых результатов на наборах данных ICDAR. Однако такие решения остаются чувствительными к объединённым ячейкам и сложным заголовочным структурам. Графовые модели (*Graph Neural Network, GNN*), такие как TGRNet [13] и TSRNet [14], переосмыслили задачу как восстановление графа ячеек, позволяя захватывать пространственные и логические связи между ними, однако они требуют дорогостоящей разметки ячеек для обучения. Решения, основанные на архитектуре трансформер (*Transformer*), в частности, TSRFormer [15], TableFormer [16], VAST [17], TATR [18] и TableVLM [19] интегрируют механизмы внимания для поэтапного декодирования координат ячеек и логической структуры, улучшая точность на сложных макетах таблиц, но по-прежнему испытывают сложности с многоуровневыми заголовками. Гибридные методы, представленные в TaPaS [20] и TaBERT [21], используют предварительное обучение на парах «*текст-таблица*» для семантического понимания содержимого столбцов и заголовков, однако их качество распознавания ухудшается на предметных областях без схожих метаданных. Системы по типу Global Table Extractor (GTE) [22] объединяют детекторы объектов с иерархическими классификаторами ячеек для совместной оптимизации определения и распознавания структуры, демонстрируя рост точности сегментации, но остаются уязвимыми к размытым границам таблиц. При этом каскадная природа этих конвейеров (*pipelines*) приводит к накоплению ошибок – промахи на этапе обнаружения таблиц усугубляются на этапе распознавания структуры. Попытки интеграции визуальных, текстовых и макетных (*layout*) признаков способствуют снижению влияния различных ошибок, но пока не существует единой архитектуры, полноценно объединяющей все модальности. Высокая вычислительная стоимость решений на основе трансформеров ограничивает их масштабирование на большие корпуса разнообразных документов с таблицами, чему противостоят исследования в направлении лёгких и адаптивных моделей для эффективного извлечения структуры таблиц, включая заголовков. Наконец, подходы вроде TSR-DSAW [23] показали, что моделирование пространственных ассоциаций слов между ячейками может улучшить обнаружение заголовков, но сильно зависит от качества идентификации отдельных слов в изображениях таблиц низкого разрешения.

Отдельно стоит упомянуть подходы, основанные на больших языковых моделях (*large language models*), которые все чаще применяются для решения различных табличных задач с использованием контекстного обучения. Представление текстовой подсказки (*prompt*) для таблицы может играть важную роль в способности моделей обрабатывать табличные данные. В частности, в работах [24-25] исследуется возможность больших языковых моделей понимать структуру таблиц, обладающих разной компоновкой, применяя технику формирования текстовых подсказок (промпт-инжиниринга – *prompt engineering*). Кроме того дается оценка производительности по отдельным форматам представления таблиц и влияния шума в данных. Однако такие походы направлены только на такие структурные задачи,

которые относятся к идентификации строк, столбцов и ячеек по заданным индексам, опуская обработку заголовков.

Таким образом, в современном ландшафте исследований по распознаванию структуры таблиц наблюдается переход от классических правил и эвристик к сложным гибридным и нейросетевым решениям, однако ни один из подходов пока не решает проблему извлечения и семантического понимания заголовков во всём их многообразии. Ключевые направления включают методы для учёта логической структуры на основе CNN-моделей, GNN-моделей и моделей на основе архитектуры трансформер, каждый из которых демонстрирует улучшения на стандартных тестовых наборах данных (бенчмарках), но испытывает трудности с обобщением на реальных таблицах со сложными иерархическими заголовками. Поэтому разработка нового методологического и программного обеспечения, решающего указанные проблемы, остаётся актуальной.

3. Методология

3.1 Формальная постановка задачи

Таблицы представляют собой структурированную форму организации данных, где информация располагается в виде сетки из строк и столбцов. Формально, исходную таблицу *T* можно представить в виде:

$$T = \langle C, A, M \rangle, \tag{1}$$

где $C = \{c_{1,1}, c_{1,2}, \dots, c_{i,j}, \dots, c_{n,m}\}$ – матрица ячеек размерности в *n*-строк и *m*-столбцов; $c_{i,j}$ – содержимое ячейки в *i*-строке *j*-столбца; $A = \{a_{i,j}\}, a_{i,j} \in \{header, data, merged\}$ – матрица структурных атрибутов (типов ячейки), где «*header*» – ячейка заголовка, «*data*» – ячейка данных «*merged*» – объединённая ячейка, при этом $rowspan(c_{i,j})$ и $colspan(c_{i,j})$ – параметры объединённых ячеек; *M* – остальные метаданные, кроме заголовков (например, подпись таблицы, текст окружения в документе, сноски). Следует отметить, что в данной работе в качестве значений ячеек рассматриваются только текстовые данные разных форматов, исключая изображения и другие подтаблицы. Метаданные *M* также не рассматриваются.

Заголовки – это специальные ячейки, которые обозначают категории или описания для данных в соответствующих строках или столбцах *T* (например, «*Год*», «*Население (млн.)*», «*Всего*»). Заголовки обычно находятся в верхней части столбцов или в начале строк, но в сложных таблицах могут располагаться в других позициях, включая многоуровневые или иерархические структуры. Формально, заголовки таблицы *H* можно представить в виде:

$$H = \{h_1, h_2, \dots, h_k\}, \tag{2}$$

где $H \subset C$ и каждый $h_l \in H, l = \overline{1,k}$ – соответствует некоторому логическому уровню иерархии (основной заголовок, подзаголовок столбцов/строк), учитывая структурные особенности *T*.

Задача автоматического распознавания заголовков *H* заключается в определении ячеек, выполняющих функцию заголовков, на основе их содержимого, форматирования и расположения в таблице. Формально это можно представить в виде отображения:

$$F(P_i): T \to H, i = \overline{1,6}, \tag{3}$$

где *F* – модель, предсказывающая заголовки (в данном исследовании используются большие языковые модели для решения этой задачи); *P* – текстовая подсказка, которая на входе получает модель (в данном исследовании предлагается шесть видов текстовых подсказок), при этом: $P_i = \{T', I\}$, где T' – текстовое представление таблицы *T*; *I* – инструкции по распознаванию заголовков *H*.

Таким образом, модель должна определить координаты или идентификаторы ячеек, являющихся заголовками, анализируя семантику, структуру и контекст данных. Ожидается, что большие языковые модели смогут обрабатывать таблицы различной сложности, включая стандартные таблицы с заголовками в первой строке, а также таблицы с многоуровневыми заголовками, объединёнными ячейками или нестандартным расположением заголовков. На рис. 1 представлена обобщенная схема дизайна предлагаемого исследования. Далее подробнее рассмотрим его основные аспекты.

Рис. 1. Общая схема дизайна предлагаемого исследования.
Fig. 1. The general design scheme of the proposed study.

3.2 Вопросы исследования

Исследование направлено на изучение факторов, влияющих на производительность больших языковых моделей в задаче распознавания заголовков таблиц. В частности, сформулированы следующие аспекты:

1) *Влияние текстовых подсказок:* Как сложность и детализация текстовой подсказки влияет на способность модели точно определять заголовки? В исследовании используются различные стратегии формирования текстовых подсказок: от простых (*zero-shot*) до сложных текстовых подсказок с пошаговыми инструкциями и примерами (*few-shot*). Улучшают ли более подробные инструкции и примеры производительность моделей?
2) *Влияние размера модели:* Как количество параметров модели влияет на точность распознавания заголовков таблиц? Ожидается ли, что более крупные модели, содержащие сотни миллиардов параметров, будут стабильно превосходить меньшие модели, или существует точка, после которой увеличение размера не приводит к значительному улучшению?
3) *Влияние структуры таблицы:* Как структурная сложность таблиц (например, наличие объединённых ячеек, многоуровневых заголовков или нестандартного расположения заголовков) влияет на точность распознавания? Способны ли модели эффективно обрабатывать сложные таблицы, или их производительность снижается при увеличении сложности структуры?

Данные вопросы охватывают ключевые переменные исследования, такие как: *вид текстовой подсказки*, *размер модели* и *структура таблиц*. Результаты позволят определить оптимальные конфигурации моделей и текстовых подсказок для различных сценариев, а также выявить ограничения больших языковых моделей в обработке сложных табличных данных.

3.3 Используемые модели

Для изучения способности больших языковых моделей распознавать заголовки таблиц выбрано восемь моделей, различающихся по размеру (количеству параметров). Модели разделены на три категории: *малые* (модели > 70 млрд. параметров), *средние* (модели < 70 и > 100 млрд. параметров) и *крупные* (модели < 100 млрд. параметров), что позволяет оценить влияние размера модели на производительность. Все модели являются инструктивно-обученными (*instruction-tuned*), что делает их подходящими для выполнения задач, требующих следования сложным инструкциям. В табл. 1 приведен список моделей с их краткими характеристиками.

Табл. 1. Большие языковые модели, используемые для исследования.
Table 1. Large language models used for research.

Название	Количество параметров	Краткое описание
Малые модели		
Mistral-7B-Instruct-v0.3 [26]	7 млрд.	Модель от Mistral AI, оптимизированная для выполнения инструкций.
Llama-3.1-8B-Instruct [27]	8 млрд.	Модель из семейства Llama 3 от Meta, предназначенная для диалогов и инструкций.
Mistral-Small-24B-Instruct-2501 [28]	24 млрд.	Более крупная модель от Mistral AI, обладающая компактным размером для запуска на одном GPU RTX 4090.
Gemma-2-27B-it [29]	27 млрд.	Модель от Google, разработанная для обработки инструктивных запросов.
Средние модели		
Llama-3.3-70B-Instruct-Turbo [30]	70 млрд.	Более крупная модель Llama 3 от Meta с улучшенными возможностями.
Qwen2-72B-Instruct [31]	72 млрд.	Модель от Qwen, оптимизированная для сложных инструктивных задач.
DeepSeek-R1-Distill-Llama-70B [32]	70 млрд.	Более эффективная и производительная дистиллированная версия Llama от DeepSeek AI.
Крупные модели		
Llama-3.1-405B-Instruct [33]	405 млрд.	Самая крупная модель Llama 3.1 от Meta с высокой производительностью.

3.4 Подготовка набора данных

Для проведения исследования выбран англоязычный набор данных PubTables-1M [34], который содержит около миллиона таблиц, извлечённых из научных статей, доступных в архиве PubMed Open Access. Этот набор данных предоставляет подробные аннотации для задач обнаружения таблиц, распознавания их структуры и функционального анализа, включая информацию о расположении ячеек, их содержимом и ролях (например, заголовки, данные). PubTables-1M был выбран благодаря его разнообразию таблиц, включающих как простые структуры с заголовками в первой строке, так и сложные таблицы с нестандартным расположением заголовков, которые могут быть многоуровневыми и иерархическими, а

также содержать объединённые ячейки. Все это делает данный набор идеальным для тестирования способности больших языковых моделей распознавать заголовки.

Этапы подготовки набора данных:

1) *Выбор таблиц.* Из исходного набора данных PubTables-1M случайным образом отобрано 237 таблиц, чтобы обеспечить репрезентативность тестового набора. Выборка охватывает таблицы различной сложности, включая стандартные таблицы с заголовками в первой строке, таблицы с иерархическими заголовками и таблицы с нестандартным расположением заголовков. Это позволяет оценить производительность моделей в разнообразных сценариях.
2) *Извлечение и очистка таблиц.* Для каждой выбранной таблицы были извлечены данные и аннотации из оригинальных JSON-файлов, предоставленных в PubTables-1M. Полные аннотации PubTables-1M включают множество деталей, таких как координаты слов, визуальные характеристики (например, размер шрифта) и ограничивающие рамки (*bounding boxes*) в формате PDF и изображений. Эти избыточные данные были удалены из выбранных таблиц. Таким образом, итоговые JSON-аннотации содержат информацию только о структуре таблицы, включая координаты ячеек, их содержимое и роли (например, является ли ячейка заголовком).
3) *Преобразование таблиц в формат DataFrame*. Согласно работе [24], большие языковые модели при решении различных структурных табличных задач лучше всего понимают формат DataFrame (JSON-подобный формат, используемый в библиотеке pandas), который показал самую высокую эффективность в экспериментах. Таким образом, принято решение преобразовать отобранные таблицы и JSON-аннотации в формат pandas DataFrame, что также упрощает последующую обработку и передачу данных в модели.
4) *Разметка таблиц.* Для каждой таблицы был сгенерирован соответствующий файл с разметкой заголовков (*ground truth*), взятых из JSON-аннотации PubTables-1M. Полученные файлы с разметкой содержат строку, указывающую на ячейки заголовков в формате: *«(row-1, column-1); (row-2, column-2); ...»*. Координаты представляют собой пары (строка, столбец), что позволяет точно идентифицировать заголовки для оценки производительности моделей.

Таким образом, подготовленный набор данных состоит из 237 таблиц в формате pandas DataFrame с соответствующими аннотациями заголовков. Данный набор предоставляет прочную основу для тестирования и сравнения различных больших языковых моделей в задаче распознавания заголовков таблиц, а также позволяет проводить анализ влияния структурной сложности таблиц на итоговый результат.

3.5 Формирование текстовых подсказок для моделей

Для взаимодействия с большими языковыми моделями разработаны шесть различных стратегий формирования текстовых подсказок, варьирующихся по уровню детализации и количеству предоставляемых примеров. Целью является оценка того, как сложность текстовой подсказки и наличие примеров влияют на способность моделей точно определять заголовки таблиц. Текстовые подсказки созданы с учётом рекомендаций, предложенных в [35], подчеркивающих важность структурированных инструкций, критериев и примеров (*few-shot*) для задач анализа таблиц.

Шесть основных стратегий формирования текстовых подсказок:

1) Простая текстовая подсказка (*zero-shot*) – представляет собой минимальный запрос, проверяющий базовую способность модели распознавать заголовки без дополнительной информации или примеров. Он подходит для оценки начальных

возможностей моделей, но может быть менее эффективен для анализа сложных таблиц. Пример текстовой подсказки:

- Пользовательский запрос: «*Ты эксперт по анализу таблиц. Определи заголовки в таблице:* {*таблица*}»

2) Текстовая подсказка с пошаговой инструкцией (*step-by-step*) – предоставляет структурированный подход, направляя модель к системному анализу таблицы, что может улучшить точность распознавания. Пример текстовой подсказки:
   - Пользовательский запрос: «*Ты эксперт по анализу таблиц. Определи заголовки в таблице:* {*таблица*}»
   - Системные инструкции: «*Подробный алгоритм, включающий следующие шаги:*
     - *Визуальный осмотр таблицы для изучения её структуры.*
     - *Идентификация потенциальных заголовков на основе обобщающего текста и форматирования.*
     - *Анализ позиционирования заголовков (сверху, слева, справа).*
     - *Формирование JSON-ответа с координатами заголовков».*
3) Текстовая подсказка с пошаговой инструкцией и критериями поиска – углубляет инструкции, предоставляя модели чёткие критерии для идентификации заголовков, что особенно полезно для сложных таблиц. Пример текстовой подсказки:
   - Пользовательский запрос: «*Ты специалист по анализу сложных таблиц. Твоя задача — определить все заголовки в представленной таблице:* {*таблица*}»
   - Системные инструкции: «*Критерии поиска заголовков:*
     - *Семантический анализ для поиска ключевых слов, указывающих на категории.*
     - *Позиционный анализ для определения расположения заголовков.*
     - *Анализ структурных паттернов (например, повторяющиеся заголовки).*
     - *Контекстуальный анализ связей между ячейками.*
     - *Анализ метаданных (размер шрифта, выравнивание).*
     - *Иерархический анализ для выделения многоуровневых заголовков».*
4) Текстовая подсказка с пошаговой инструкцией и критериями, включающая один пример (*1-shot*) – данная текстовая подсказка аналогична предыдущей, но включает также один пример таблицы с её правильными аннотациями заголовков. Пример содержит входные данные (таблицу в формате pandas DataFrame) и ожидаемый выход (JSON с координатами и характеристиками заголовков, такими как текст, позиция и уровень иерархии). Пример помогает модели понять ожидаемый формат ответа и тип заголовков, что может улучшить производительность.
5) Текстовая подсказка с пошаговой инструкцией и критериями, включающая два примера (*2-shot*) – включает два примера таблиц с аннотациями заголовков, охватывающих разные структуры (например, заголовки сверху и сбоку). Это увеличивает контекст и помогает модели лучше обобщать информацию.
6) Текстовая подсказка с пошаговой инструкцией и критериями, включающая три примера (*3-shot*) – наиболее подробная текстовая подсказка, содержащая три примера таблиц с аннотациями. Примеры включают таблицы с различными структурами, такими как многоуровневые заголовки и нестандартное расположение, предоставляя модели максимальный контекст для обучения на образцах.

Каждая текстовая подсказка была разработана для оценки влияния уровня детализации и количества примеров на точность распознавания заголовков. Так простая текстовая

подсказка (*zero-shot*) тестирует базовые способности модели, тогда как текстовые подсказки с примерами (*few-shot*) проверяют, насколько модель может использовать дополнительный контекст для улучшения результатов. Это позволяет определить оптимальную стратегию формирования текстовых подсказок для различных моделей и типов таблиц. Однако увеличение сложности текстовой подсказки может также увеличить вычислительные затраты и требования к контекстному окну модели.

Табл. 2 суммирует характеристики текстовых подсказок и их предполагаемую эффективность, основанную на уровне детализации и наличии примеров.

Табл. 2. Основные характеристики стратегий формирования текстовых подсказок. «int» – индикатор пошаговых инструкций; «crit» – индикатор критериев поиска.
Table 2. The main characteristics of prompting strategies: "int" is an indicator of step-by-step instructions; "crit" is an indicator of search criteria.

№	Вид текстовой подсказки	Пошаговый алгоритм	Наличие критериев поиска	Наличие примеров	Ожидаемая эффективность
1	*zero-shot*	–	–	–	*низкая*
2	*zero-shot+int*	+	–	–	*средняя*
3	*zero-shot+int+crit*	+	+	–	*выше среднего*
4	*1-shot+int+crit*	+	+	+	*высокая*
5	*2-shot+int+crit*	+	+	+	*высокая*
6	*3-shot+int+crit*	+	+	+	*очень высокая*

4. Эксперименты

4.1 Экспериментальные настройки

Для автоматизации проведения экспериментов по распознаванию координат ячеек с заголовками в табличных данных была разработана специальная программная среда на языке Python 3.10. При этом использовалась библиотека LangChain [36], в частности, плагин langchain_together, обеспечивающего удобный интерфейс взаимодействия с API платформы Together.ai [37]. Все запросы к языковым моделям производились через данный API с температурой генерации (*temperature*) равной 0.0 для обеспечения стабильности и воспроизводимости результатов (ответов).

Отобранные и подготовленные таблицы для экспериментов хранятся в каталоге «*Tables_for_models*», каждая из которых сопровождалась отдельным файлом с координатами ячеек, размеченных как заголовки в формате *row_index* и *col_index*.

С использованием разработанной среды было протестировано 8 языковых моделей (модели типа LLaMA, Mistral, DeepSeek, Gemma и Qwen) в сочетании с 6 различными стратегиями текстовых подсказок. В итоге общее количество запросов составило: 237 таблиц × 6 текстовых подсказок × 8 моделей = 11 376 запросов.

4.2 Метрики оценки

В качестве основных метрик оценки качества распознавания заголовков использовались стандартные метрики: *точности* (*Precision*), *полноты* (*Recall*) и *F-меры* (*F1*).

$$Precision = \frac{|TP|}{|TP|+|FP|}, \quad Recall = \frac{|TP|}{|TP|+|FN|}, \quad F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}, \tag{4}$$

где *TP (True Positives)* – корректно предсказанные координаты заголовков моделью; *FP (False Positives)* – координаты, предсказанные моделью, но не являющиеся заголовками; *FN (False Negatives)* – координаты истинных заголовков, не найденные моделью.

4.3 Результаты и обсуждение

Последовательная обработка 11 376 запросов заняла ≈ 49 344 секунд (≈ 13,7 часов). Переход на параллельную обработку в пять потоков снизил это время до ≈ 9 869 секунд (≈ 2,7 часов), что ускорило выполнение обработки запросов почти в пять раз.

Результаты экспериментальной оценки представлены на рис. 2-4. Они демонстрируют тепловую карту значений полученных оценок точности, полноты и F-меры для каждой комбинации «*модель* × *текстовая подсказка*», где на оси «*Y*» представлены названия моделей, а оси «*X*» – номера текстовых подсказок (от самой простой «*zero-shot*» 1 до самой сложной «*3-shot+int+crit*» 6). При этом тёмно-синие цвета соответствуют высоким оценкам, например, F1 (≈ 0.80–0.85) и сконцентрированы в строках крупных и средних моделей, таких как Meta-Llama-3.1-405B и Llama-3.3-70B. У моделей Gemma-2-27b-it и DeepSeek-R1-Distill-Llama-70B преобладают средние значения (оттенки синего с примесью зелёного), что указывает на оценки F1 около 0.70–0.80. Более светлые, «приглушённые» участки в нижней части карты – это результаты малых моделей, таких как Mistral-7B и Mistral-Small-24B, где оценки F1 падает до 0.06–0.30 в зависимости от текстовых подсказок.

Первые две подсказки (номера 1 и 2) дают максимально выраженный контраст: крупные и средние модели уверенно работают в любом режиме, тогда как мелкие сильно реагируют на усложнённую текстовую подсказку под номером 1. Столбцы с текстовыми подсказками 3–6 менее вариативны для больших моделей, что свидетельствует об их устойчивости к формулировке задания, в то время как у малых моделей цвета ряда ещё светлее, показывая деградацию качества.

На рис. 5 приведена радиальная диаграмма, которая представляет собой визуализацию значений оценок F-меры для различных моделей в зависимости от текстовых подсказок. Каждая ось диаграммы соответствует конкретной текстовой подсказки (от 1 до 6), а линии или области обозначают модели разного масштаба.

В целом данная диаграмма соотносится с тепловой картой, представленной на рис. 4. Так, на диаграмме крупные и средние модели, такие как Meta-Llama-3.1-405B и Llama-3.3-70B, образуют обширные многоугольники с вершинами, значительно удалёнными от центра. Это указывает на высокие значения оценки F1 (примерно 0.80–0.85) по всем текстовым подсказкам. Их области занимают большую часть диаграммы, подчёркивая превосходство этих моделей в решении задач независимо от их сложности. Модели среднего уровня, такие как Gemma-2-27b-it и DeepSeek-R1-Distill-Llama-70B, формируют многоугольники с умеренным радиусом. Их вершины расположены на среднем расстоянии от центра, что соответствует оценки F1 в диапазоне 0.70–0.80. Это отражает стабильную, но не выдающуюся производительность. Малые модели, такие как Mistral-7B и Mistral-Small-24B, представлены многоугольниками, стянутыми к центру диаграммы. Их вершины находятся близко к началу координат, что свидетельствует о низких значениях F1-меры (0.06–0.30), особенно на некоторых текстовых подсказках.

При рассмотрении отдельных осей видно, что на текстовых подсказках 1 и 2 крупные и средние модели демонстрируют максимальные радиусы, тогда как малые модели едва выходят за пределы центральной области. Это подчёркивает слабую адаптацию малых моделей к усложнённым заданиям, особенно на текстовой подсказке 2. На текстовых подсказках 3–6 крупные и средние модели сохраняют стабильно высокие радиусы, что указывает на их устойчивость к вариациям в формулировке задач. Малые модели, напротив, продолжают показывать низкие значения, их многоугольники остаются сжатыми, что говорит о снижении качества при обработке более сложных текстовых подсказок.

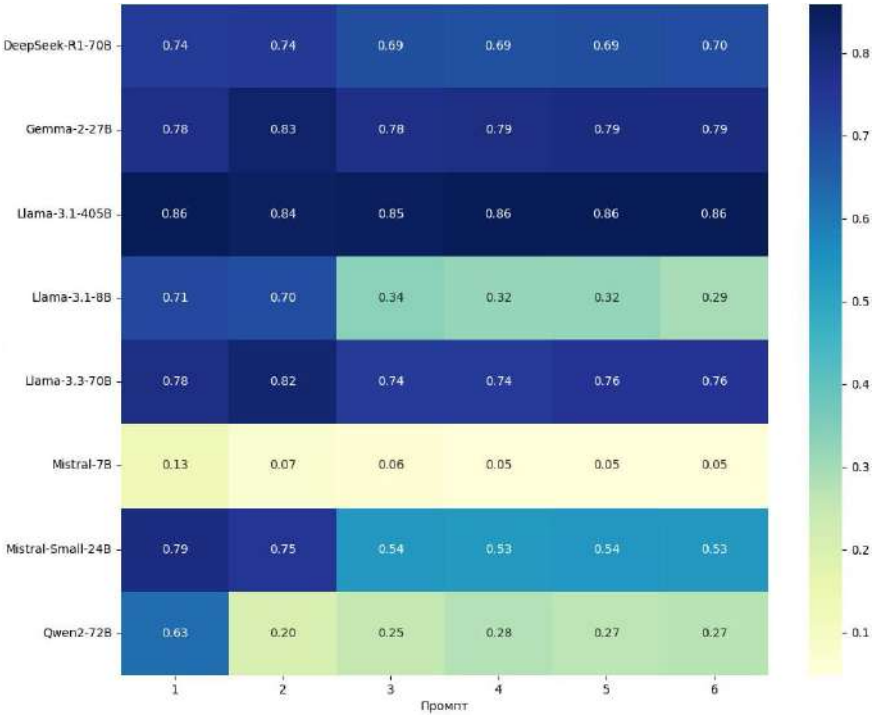


Рис. 2. Тепловая карта полученных значений оценки точности.
Fig. 2. Heat map of the obtained precision values.

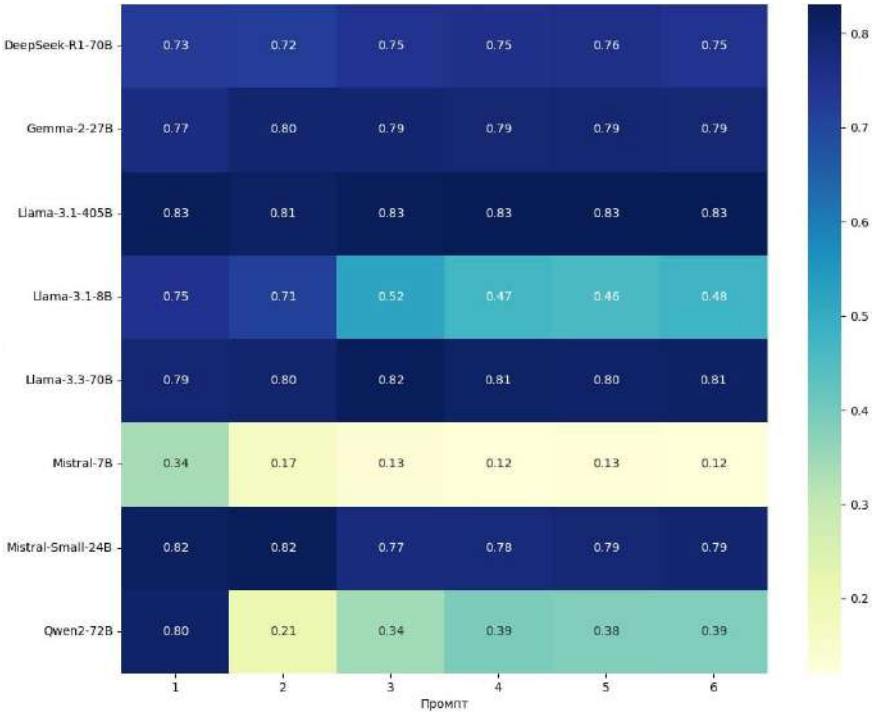


Рис. 3. Тепловая карта полученных значений оценки полноты.
Fig. 3. Heat map of the obtained recall values.

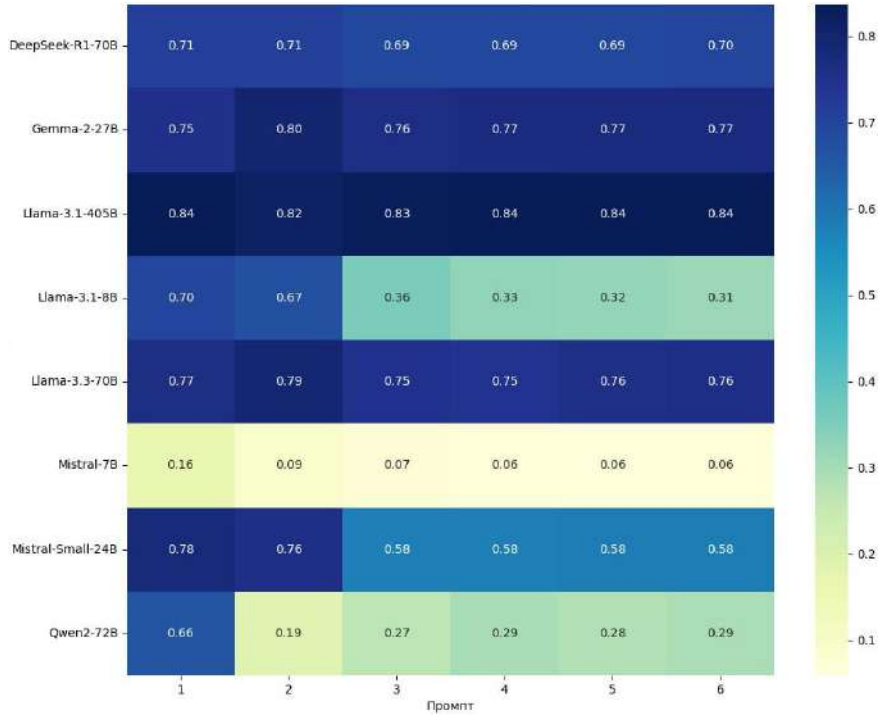


Рис. 4. Тепловая карта полученных значений оценки F-меры.
Fig. 4. Heat map of the obtained F1 score values.

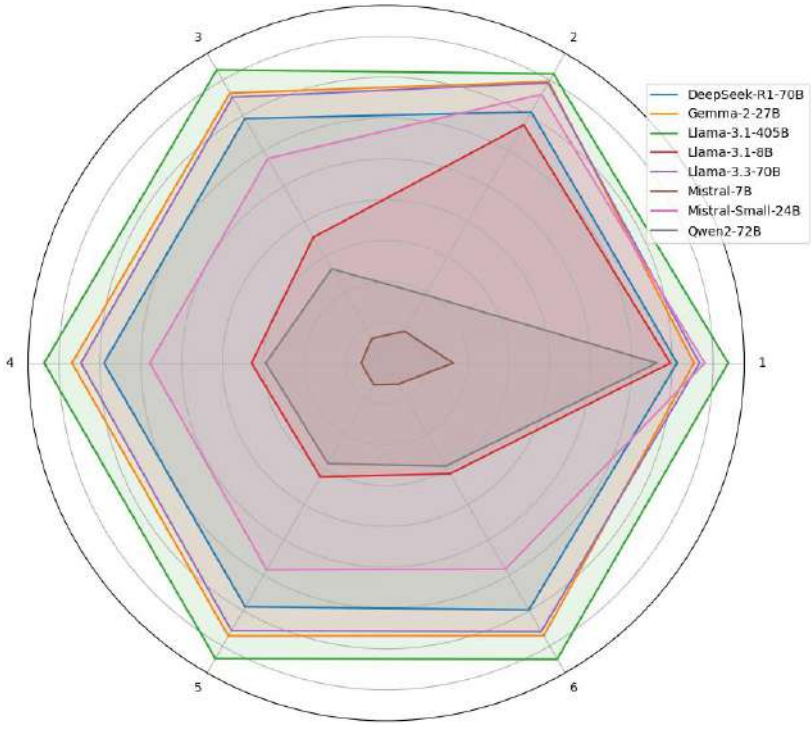


Рис. 5. Радиальная диаграмма полученных значений оценки F-меры.
Fig. 5. Radial diagram of the obtained F1 score values.

Проведенные эксперименты показали, что модели Meta-Llama-3.1-8B, Qwen2-72B и Mistral-7B демонстрируют наибольшую нестабильность при применении усложнённых текстовых подсказок. Вместо ожидаемого улучшения качества посредствам пошаговых инструкций, критериев поиска или несколькими примерами (*few-shot*), у этих моделей наблюдалось:

- **Негативное влияние многосоставных текстовых подсказок:** сложные инструкции иногда вводили модель в заблуждение, при этом общая F-мера снижалась.
- **Нестандартный формат ответов:** несмотря на явные указания в текстовых подсказках о выдаче координат заголовков, модели часто возвращали не числа (координаты), а содержимое ячеек. Вероятно, это связано с архитектурной спецификой генеративных моделей на основе трансформеров, которые оптимизированы на предсказание следующих токенов, а не числовых координат, поэтому они извлекают и возвращают текстовую часть (заголовок), а не позицию. Другими словами модель не получает достоверного сигнала о геометрической позиции и вместо этого использует наиболее вероятный паттерн в виде текста предполагаемого заголовка. Также может быть вариант, что большие языковые модели не специализируются на этой задаче и при их обучении использовались базовые таблицы с заголовком в первой строке, в связи с чем, они склонны использовать этот паттерн.
- **Необходимость адаптивного парсинга ответов:** чтобы компенсировать неконсистентные ответы, был реализован дополнительный шаг – при получении текстовых значений осуществляется их поиск в исходной таблице и сопоставление с координатами. Эта правка позволила исправить одну из самых распространённых ошибок и отразилась на итоговых оценках.

Основным выводом данного эксперимента является то, что общая ожидаемая тенденция «*чем сложнее текстовая подсказка, тем лучше результат*» не оправдала ожиданий. Скорее, наоборот: у мелких моделей избыточная детализация подсказки вызывала ухудшение распознавания, они «путались» в инструкциях. У средних и крупных моделей зависимость от сложности подсказки была менее выражена – их F-мера оставалась относительно стабильной. Таким образом, простые, чёткие текстовые подсказки оказались эффективнее для большинства моделей.

Было отмечено, что модели сильно теряли производительность, если заголовки находились не на первой строке:

- Для средних и сложных таблиц (многоуровневых иерархических заголовков с объединёнными ячейками) практически все модели показывали низкую оценку полноты, пропуская истинные заголовки, расположенные вне первой строки.
- Крупные и средние модели справлялись с такими случаями лучше, но всё равно демонстрировали заметный спад F-меры по сравнению с простыми таблицами.

На данный момент анализ основан на 237 таблицах, из которых лишь небольшая часть (20 таблиц) относится к сложным структурам. Поэтому полученные оценки в большей степени отражают поведение моделей на таблицах с более простой структурой. Для более точной оценки способности моделей работать с нетипичными макетами таблиц требуется расширить выборку за счёт увеличения числа сложных таблиц.

В целом, полученные результаты показывают, что:

- *Размер параметров модели* по-прежнему остаётся главным фактором качества.
- *Чёткость и минимализм текстовых подсказок* критичны даже для средних и крупных моделей.
- *Адаптивный парсинг ответов* обязательный шаг при использовании различных

больших языковых моделей.

- Полностью оценить *устойчивость к сложным таблицам* можно лишь после расширения выборки.

5. Заключение

В данной работе исследуются возможности больших языковых моделей распознавать заголовки таблиц, обладающих разной структурной компоновкой. Эксперименты проведены на выборке из 237 таблиц, отобранных на основе крупномасштабного корпуса PubTables-1M. Проведенное исследование демонстрирует, что модели способны решать данную задачу, но их производительность существенно варьируется в зависимости от размера модели и стратегии формирования текстовых подсказок. Крупные модели (например, Meta-Llama-3.1-405B) показывают стабильно высокие результаты (F1 ≈ 0.80–0.85), тогда как малые модели (например, Mistral-7B) значительно отстают (F1 ≈ 0.06–0.30). Усложнение текстовых подсказок за счет пошаговых инструкций, критериев поиска и примеров (*few-shot*) не всегда улучшает качество, в частности, для малых моделей это приводит к деградации результатов. Наибольшие сложности возникают при обработке таблиц с многоуровневыми заголовками и объединёнными ячейками, где даже у крупных и средних моделей снижается точность распознавания.

В будущей работе планируется расширить собранный набор данных, включив примеры таблиц, обладающих более сложной структурой, а также примеры с наличием различных шумов (ошибок, опечаток и т.п.), присутствующих изначально в выбранном корпусе PubTables-1M. Однако следует отметить, что набор данных составлен на английском языке, так как оригинальные таблицы и аннотации PubTables-1M основаны на англоязычных научных статьях. Поэтому планируется создать аналогичный набор данных на русском языке путём перевода основного набора и сбора новых таблиц, например, из русскоязычной части Википедии или GitHub. Это позволит оценить влияние языка на производительность моделей и их способность обрабатывать таблицы на разных языках. Кроме того, предлагается исследовать гибридные подходы, комбинирующие большие языковые модели с традиционными методами компьютерного зрения, для улучшения обработки визуальных особенностей таблиц. Эти шаги позволят создать более надежные системы автоматического анализа заголовков таблиц, применимые в реальных сценариях, таких как интеграция с базами данных и поддержка научных исследований.

Список литературы

[1]. Dong H., Cheng Z., He X., Zhou M., Zhou A., Zhou F., Liu A., Han S., Zhang D. Table Pre-training: A Survey on Model Architectures, Pre-training Objectives, and Downstream Tasks. Proc. the Thirty-First International Joint Conference on Artificial Intelligence, Vienna, Austria, 2022, pp. 5426-5435. DOI: 10.24963/ijcai.2022/761.

[2]. Badaro G., Saeed M., Papotti P. Transformers for Tabular Data Representation: A Survey of Models and Applications. Transactions of the Association for Computational Linguistics, vol. 11, 2023, pp. 227-249. DOI: 10.1162/tacl_a_00544.

[3]. Dong H., Wang Z. Large Language Models for Tabular Data: Progresses and Future Directions. Proc. the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'24), Washington, USA, 2024, pp. 2997-3000. DOI: 10.1145/3626772.3661384.

[4]. Bonfitto S., Casiraghi E., Mesiti M. Table understanding approaches for extracting knowledge from heterogeneous tables. Wiley Interdisciplinary Reviews, Data Mining and Knowledge Discovery, vol. 11, 2021, e1407. DOI: 10.1002/widm.1407.

[5]. Shigarov A. Table understanding: Problem overview. Wiley Interdisciplinary Reviews, Data Mining and Knowledge Discovery, vol. 13, 2022, e1482. DOI: 10.1002/widm.1482.

[6]. Embley D. W., Krishnamoorthy M. S., Nagy G., Seth S. Converting heterogeneous statistical tables on the web to searchable databases. International Journal on Document Analysis and Recognition, vol. 19, no. 2, 2016, pp. 119-138. DOI: 10.1007/s10032-016-0259-1.

[7]. Rastan R., Paik H.-Y., Shepherd J. TEXUS: a unified framework for extracting and understanding tables in PDF documents. Information Processing and Management: an International Journal, vol. 56, no. 3, 2019, pp. 895-918. DOI: 10.1016/j.ipm.2019.01.008.
[8]. Wu X., Chen H., Bu C., Ji S., Zhang Z., Sheng V. S. HUSS: A heuristic method for understanding the semantic structure of spreadsheets. Data Intelligence, vol. 5, no. 3, 2023, pp. 537-559. DOI: 10.1162/dint_a_00201.
[9]. Roldán J. C., Jiménez P., Corchuelo R. On extracting data from tables that are encoded using html. Knowledge-Based Systems, vol. 190, 2020, 105157. DOI: 10.1016/j.knosys.2019.105157.
[10]. Fang J., Mitra P., Tang Z., Giles C. L. Table header detection and classification. Proc. the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI'12), Toronto, Ontario, Canada, 2012, pp. 599-605. DOI: 10.5555/2900728.2900814.
[11]. Roldán J. C., Jiménez P., Szekely P., Corchuelo R. TOMATE: A heuristic-based approach to extract data from HTML tables. Information Sciences, vol. 577, 2021, pp. 49-68. DOI: 10.1016/j.ins.2021.04.087.
[12]. Fetahu B., Anand A., Koutraki M. TableNet: An Approach for Determining Fine-grained Relations for Wikipedia Tables. Proc. the World Wide Web Conference (WWW'19), San Francisco, CA, USA, 2019, pp. 2736-2742. DOI: 10.1145/3308558.3313629.
[13]. Xue W., Yu B., Wang W., Tao D., Li Q. TGRNet: A Table Graph Reconstruction Network for Table Structure Recognition. Proc. 2021 IEEE/CVF International Conference on Computer Vision (ICCV), Montreal, QC, Canada, 2021, pp. 1295-1304. DOI: 10.1109/ICCV48922.2021.00133.
[14]. Li X. H., Yin F., Dai H. S., Cheng-Lin Liu C. L. Table Structure Recognition and Form Parsing by End-to-End Object Detection and Relation Parsing. Pattern Recognition, vol. 132, no. C, 2022, DOI: 10.1016/j.patcog.2022.108946.
[15]. Lin W., Sun Z., Ma C., Li M., Wang J., Sun L., Huo Q. TSRFormer: Table Structure Recognition with Transformers. Proc. the 30th ACM International Conference on Multimedia (MM'22), New York, USA, 2022, pp. 6473-6482. DOI: 10.1145/3503161.3548038.
[16]. Yang J., Gupta A., Upadhyay S., He L., Goel R., Paul S. TableFormer: Robust Transformer Modeling for Table-Text Encoding. Proc. the 60th Annual Meeting of the Association for Computational Linguistics (ACL'22), Dublin, Ireland, 2022, pp. 528-537. DOI: 10.18653/v1/2022.acl-long.40.
[17]. Huang Y., Lu N., Chen D., Li Y., Xie Z., Zhu S., Gao L., Peng W. Improving Table Structure Recognition with Visual-Alignment Sequential Coordinate Modeling. Proc. 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Vancouver, BC, Canada, 2023, pp. 11134-11143. DOI: 10.1109/CVPR52729.2023.01071.
[18]. Smock B., Pesala R., Abraham R. Aligning Benchmark Datasets for Table Structure Recognition. Proc. 17th International Conference of Document Analysis and Recognition (ICDAR'2023), San Jose, CA, USA, 2023, pp 371-386. DOI: 10.1007/978-3-031-41734-4_23.
[19]. Chen L., Huang C., Zheng X., Lin J., Huang X. TableVLM: Multi-modal Pre-training for Table Structure Recognition. Proc. the 61st Annual Meeting of the Association for Computational Linguistics (ACL'23), Toronto, Canada, 2023, pp. 2437-2449. DOI: 10.18653/v1/2023.acl-long.137.
[20]. Herzig J., Nowak P. K., Muller T., Piccinno F., Eisenschlos J. M. TaPas: Weakly Supervised Table Parsing via Pre-training. Proc. 58th Annual Meeting of the Association for Computational Linguistics, Online, 2020, pp. 4320-4333. DOI: 10.18653/v1/2020.acl-main.398.
[21]. Yin P., Neubig G., Yih W. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. Proc. the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 8413-8426. DOI: 10.18653/v1/2020.acl-main.745.
[22]. Zheng X., Burdick D., Popa L., Zhong X., Wang N. X. R. Global Table Extractor (GTE): A Framework for Joint Table Identification and Cell Structure Recognition Using Visual Context. Proc. 2021 IEEE Winter Conference on Applications of Computer Vision (WACV), Waikoloa, HI, USA, 2021, pp. 697-706. DOI: 10.1109/WACV48630.2021.00074.
[23]. Jain A., Paliwal S., Sharma M., Vig L. TSR-DSAW: Table Structure Recognition via Deep Spatial Association of Words. Proc. the European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN'2021), Online, 2021, pp. 257-262. DOI: 10.14428/esann/2021.es2021-109.
[24]. Singha A., Cambronero J., Gulwani S., Le V., Parnin C. Tabular Representation, Noisy Operators, and Impacts on Table Structure Understanding Tasks in LLMs. Proc. Table Representation Learning Workshop at NeurIPS 2023, Online, 2023, pp. 1-14. DOI: arxiv.org/abs/2310.10358.
[25]. Sui Y., Zhou M., Zhou M., Han S., Zhang D. Table Meets LLM: Can Large Language Models Understand Structured Table Data? A Benchmark and Empirical Study. Proc. the 17th ACM International Conference

on Web Search and Data Mining (WSDM'24), Merida, Mexico, 2024, pp. 645-654. DOI: 10.1145/3616855.3635752.
[26]. Mistral-7B-Instruct-v0.3, Available at: https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3, accessed 09.05.2025.
[27]. Llama-3.1-8B-Instruct, Available at: https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, accessed 09.05.2025.
[28]. Mistral-Small-24B-Instruct-2501, Available at: https://huggingface.co/mistralai/Mistral-Small-24B-Instruct-2501, accessed 09.05.2025.
[29]. Gemma-2-27b-it, Available at: https://huggingface.co/google/gemma-2-27b-it, accessed 09.05.2025.
[30]. Llama-3.3-70B-Instruct, Available at: https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct, accessed 09.05.2025.
[31]. Qwen2-72B-Instruct, Available at: https://huggingface.co/Qwen/Qwen2-72B-Instruct, accessed 09.05.2025.
[32]. DeepSeek-R1-Distill-Llama-70B, Available at: https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Llama-70B, accessed 09.05.2025.
[33]. Llama-3.1-405B-Instruct, Available at: https://huggingface.co/meta-llama/Llama-3.1-405B-Instruct, accessed 09.05.2025.
[34]. Smock B., Pesala R., Abraham R. PubTables-1M: Towards comprehensive table extraction from unstructured documents. Proc. 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, LA, USA, 2022, pp. 4634-4642. DOI: 10.1109/CVPR52688.2022.00459.
[35]. A developer's guide to prompt engineering and LLMs, Available at: https://github.blog/ai-and-ml/generative-ai/prompt-engineering-guide-generative-ai-llms/, accessed 09.05.2025.
[36]. LangChain framework, Available at: https://www.langchain.com/, accessed 09.05.2025.
[37]. Together AI, Available at: https://www.together.ai/, accessed 09.05.2025.

Информация об авторах / Information about authors

Илья Игоревич ОХОТИН – магистрант Института математики и информационных технологий Иркутского государственного университета (ИМИТ ИГУ) с 2024 года. Сфера научных интересов: большие языковые модели, работа с табличными данными, распознавание структуры таблиц; обработка заголовков таблиц.

Ilia Igorevich OKHOTIN is a master student at the Institute of Mathematics and Information Technology of Irkutsk State University (IMIT ISU) since 2024. Research interests: large language models, tabular data processing, table structure recognition; table header processing.

Никита Олегович ДОРОДНЫХ – кандидат технических наук, старший научный сотрудник Института динамики систем и теории управления им. В.М. Матросова Сибирского отделения РАН (ИДСТУ СО РАН) с 2021 года. Сфера научных интересов: автоматизация создания интеллектуальных систем и баз знаний, получение знаний на основе преобразования концептуальных моделей и электронных таблиц.

Nikita Olegovych DORODNYKH – Cand. Sci. (Tech.), senior associate researcher at the Matrosov Institute of System Dynamics and Control Theory named SB RAS (ISDCT SB RAS) since 2021. Research interests: computer-aided development of intelligent systems and knowledge bases, knowledge acquisition based on the transformation of conceptual models and tables.

DOI: 10.15514/ISPRAS-2025-37(6)-10

Enhancing E-Government Services through Chatbot Development Using Azure OpenAI

L. Radosavljević, ORCID: 0009-0005-6666-8913 <lr20180196@student.fon.bg.ac.rs>
M. Simić, ORCID: 0000-0002-6870-2303 <milica.simic@fon.bg.ac.rs>
A. Joksimović, ORCID: 0009-0008-5711-7636 <aleksandar.joksimovic@elab.rs>
T. Naumović, ORCID: 0000-0001-9849-7665 <tamara.naumovic@fon.bg.ac.rs>
M. Despotović-Zrakić, ORCID: 0000-0002-6458-1575
<marijana.despotovic-zrakic@fon.bg.ac.rs>

University of Belgrade – Faculty of Organizational Sciences,
Jove Ilića 154, Belgrade, 11000, Serbia.

Abstract. Implementing and developing chatbots as e-government services contributes significantly to the modernization and increased efficiency of public services. By leveraging Microsoft technologies - specifically the Azure OpenAI service - it is possible to rapidly and effectively develop intelligent chatbots. When integrated with the e-government portal, such chatbots offer users improved access to information and enable personalized communication between citizens and government institutions. A key issue currently lies in the lack of effective communication channels, which results in longer response times and reduced user satisfaction. The objective of this paper is to develop a chatbot that enhances service quality and brings public services closer to citizens via the e-government portal. The paper analyzes chatbot functionalities such as response generation, relevance checking of user queries, and information filtering. Furthermore, attention is given to legal and ethical considerations, data protection, and continuous model training to maintain data accuracy. This paper also explores and demonstrates how modern artificial intelligence technologies contribute to making public services more accessible to users. The proposed solution to the identified challenges involves the implementation of a chatbot model integrated with the e-government portal to improve communication. Ultimately, the focus is placed on the digitalization and modernization of public sector services to deliver benefits for society as a whole.

Keywords: chatbot; Azure OpenAI; e-government; artificial intelligence.

For citation: Radosavljević L., Simić M., Joksimović A., Naumović T., Despotović-Zrakić M. Enhancing e-government services through chatbot development using Azure OpenAI. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 167-180. DOI: 10.15514/ISPRAS-2025-37(6)-10.

Acknowledgements. The authors are thankful to the Ministry of Science, Technological Development, and Innovation of the Republic of Serbia for financial support through institutional funding of FON.

Улучшение электронных государственных услуг посредством разработки чат-ботов с использованием Azure OpenAI

Л. Радосавльевич, ORCID: 0009-0005-6666-8913 <lr20180196@student.fon.bg.ac.rs>
М. Симич, ORCID: 0000-0002-6870-2303 <milica.simic@fon.bg.ac.rs>
А. Йоксимович, ORCID: 0009-0008-5711-7636 <aleksandar.joksimovic@elab.rs>
Т. Наумович, ORCID: 0000-0001-9849-7665 <tamara.naumovic@fon.bg.ac.rs>
М. Деспотович-Зракич, ORCID: 0000-0002-6458-1575 <marijana.despotovic-zrakic@fon.bg.ac.rs>

Университет Белграда – Факультет Организационных наук
ул. Йове Илича, 154, Белград, 11000, Сербия.

Аннотация. Реализация и развитие чат-ботов как сервиса электронного правительства в значительной степени способствуют модернизации и повышению эффективности государственных услуг. Используя технологии Microsoft - в частности, сервис Azure OpenAI - возможно быстро и эффективно разработать интеллектуальных чат-ботов. При интеграции с порталом электронного правительства такие чат-боты обеспечивают пользователям улучшенный доступ к информации и персонализированную коммуникацию между гражданами и государственными учреждениями. Одной из ключевых проблем на сегодняшний день является отсутствие эффективных каналов связи, что приводит к увеличению времени отклика и снижению удовлетворенности пользователей. Цель данной работы – разработка чат-бота, который повысит качество предоставляемых услуг и сделает государственные сервисы более доступными гражданам через портал электронного правительства. В работе анализируются функциональные возможности чат-бота, такие как генерация ответов, проверка релевантности заданных вопросов и фильтрация информации. Кроме того, внимание уделяется правовым и этическим аспектам, защите данных, а также постоянному обучению модели для поддержания точности данных. В исследовании также рассматривается и демонстрируется, каким образом современные технологии, основанные на искусственном интеллекте, способствуют приближению государственных услуг к пользователям. Предлагаемое решение выявленных проблем заключается в внедрении модели чат-бота с интеграцией в портал электронного правительства для улучшения коммуникации. В конечном итоге основное внимание уделяется цифровизации и модернизации услуг в государственном секторе для получения выгод для общества в целом.

Ключевые слова: чат-бот; сервис Azure OpenAI; электронное правительство; искусственный интеллект.

Для цитирования: Радосавльевич Л., Симич М., Йоксимович А., Наумович Т., Деспотович-Зракич М. Улучшение электронных государственных услуг посредством разработки чат-ботов с использованием Azure OpenAI. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 167–180 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–10.

Благодарности. Авторы выражают благодарность Министерству науки, технологического развития и инноваций Республики Сербия за финансовую поддержку через институциональное финансирование ФОН.

1. Introduction

The modernization of technology and the process of digitalization have significantly transformed the way communication occurs between citizens and government institutions. In the past, users had to be physically present and wait in long queues to submit various documents. Today, each of these processes has a digital counterpart. Signing documents, obtaining certificates and official records, and paying fees can now be accomplished with significantly less time and effort. This transformation not only reduces administrative costs but also provides citizens, i.e., users, with easier access to public services.

Despite the digitalization of public services, users often encounter obstacles when attempting to utilize them. The wide range of information available on government portals, along with guidance

that is sometimes not particularly intuitive, can lead to confusion and cause citizens to abandon the digital service altogether. To overcome these challenges and facilitate communication, there is a growing need for new, modern technologies, such as chatbots.

The aim of this paper is to explore the application of chatbots in e-government in order to identify their advantages, limitations, and challenges in improving communication between governmental institutions and citizens. The main issue lies in the inefficiency of traditional methods of delivering public services, which are often slow, overly bureaucratic, and limited in accessibility. As a proposed solution, this paper analyzes how artificial intelligence-based chatbot models can automate processes, enhance communication, and simplify access to information and services.

The paper is organized into several interrelated chapters. The second chapter provides definitions of key concepts related to chatbots, an overview of their application in e-government, and a review of relevant Azure technologies used for AI system development, including Azure OpenAI, Azure App Service, Azure AI Search, and Azure Cosmos DB. The third chapter focuses on the development of a chatbot as an e-government service using the Azure OpenAI service. The fourth chapter presents the implemented solution. Finally, the conclusion offers directions for further development and potential improvements.

2. Literature Review

2.1 Chatbots

A chatbot is a computer program designed to simulate human conversation with a user. In order to understand user queries and provide appropriate feedback, it relies on artificial intelligence (AI) and natural language processing (NLP) techniques. Chatbots are widely used across various domains such as sales, education, customer support, and many others. They are commonly integrated into web portals where they serve as digital assistants, and are also frequently implemented on social media platforms and mobile applications [1].

There are several classifications of chatbots that differ in complexity, sophistication, and other criteria. However, three main types fall into the category that utilizes natural language processing (NLP), and they differ primarily in purpose [2-3]:

1) **Rule-based chatbots** – These operate based on predefined rules. They use automation to provide responses based on user input, making them less complex compared to other types.
2) **Retrieval-based chatbots** – These select a suitable response from a set of predefined replies by analyzing the user's input. They offer more flexibility than rule-based bots but still don't generate responses.
3) **Generative AI chatbots** – These are based on artificial intelligence and are capable of producing dynamic responses and understanding the broader context. However, they can sometimes generate incorrect or inaccurate answers. Due to this risk, the models must be continuously trained in order to minimize the occurrence of false or misleading outputs.

The next section provides an overview of the core technologies underlying modern chatbots: artificial intelligence, machine learning, natural language processing, and large language models. It explains how each contributes to design and operation, laying the foundations for the architectural and implementation choices that follow.

2.1.1 AI technologies for chatbot development

Chatbots, as advanced AI-based solutions, are built upon several fundamental technologies from the field of artificial intelligence. The most important among these are Artificial Intelligence (AI) itself, Machine Learning (ML), Natural Language Processing (NLP), and Large Language Models (LLMs). Each of these plays a specific role in enabling chatbots to understand, process, and generate human-like responses.

Artificial Intelligence is a comprehensive discipline within computer science that empowers machines to simulate human cognitive functions, such as perception, reasoning, and problem-solving. Those functionalities are achieved by iteratively training sophisticated models on vast and heterogeneous datasets, thereby enabling systems to continuously refine their predictive capabilities, generalize across diverse tasks, and adapt to evolving environments [4].

Machine Learning is a subfield of artificial intelligence focused on developing algorithms that learn from data. It enables computer systems to improve their performance over time by analyzing and adapting to new information. The training data can include various formats such as images, text, numerical data, or reports. This data is prepared and used to create training datasets. Generally, using high-quality data leads to better model performance and more accurate outputs [5], yet these data-driven predictions should be validated when factual precision is critical.

Natural Language Processing is a technology that enables machines to understand, interpret, and generate human language. NLP systems process large volumes of text, audio, or video data, often collected from platforms such as social media, emails, or communication tools [6]. By applying machine learning techniques, NLP allows for the automatic analysis and extraction of meaningful information from this data. NLP is frequently integrated into applications and services that require interaction with users, such as chatbots and virtual assistants. Advances in NLP have been key to the development of generative AI models capable of simulating human conversation or even creating images from text-based prompts [7].

Large Language Models are advanced AI models capable of understanding and generating human-like text based on the data they were trained on. One notable example is OpenAI's ChatGPT. These models are trained on vast datasets to recognize linguistic patterns and generate coherent responses. LLMs are built on deep neural networks, particularly the transformer architecture, which utilizes attention mechanisms to understand relationships between words and maintain context [8]. LLM performance improves with larger training datasets, although this comes with increased computational requirements. Due to their ability to generate natural-sounding language, LLMs are widely used in areas such as code generation, content creation, and language translation [9].

Together, AI, ML, NLP, and LLMs form the technological backbone of modern chatbots. Each of these technologies contributes a specific layer of capability, from data processing and pattern recognition to language understanding and generation. In the next section we will illustrate their interplay in e-government conversational agents.

2.1.2 Chatbots in e-government

One of the most prominent trends today in the private sector is the use of applications that communicate with users in a conventional, human-like manner. Private organizations operating in industries such as banking, tourism, finance, and others increasingly use these applications to automate communication and process management with clients. The first generation of chatbots had limited capabilities and could only respond to simple and clearly defined queries. However, as technology progressed, so did their functionalities - modern generations of chatbots can now even generate software code. Although the benefits of these technologies are mostly utilized in the private sector, more and more public sector organizations are starting to adopt AI-based systems [10].

In addition to the numerous benefits these models bring, their application also raises new questions and challenges. One such challenge is information extraction. This requires the development of a high-quality knowledge base and the preservation of data relevance. In the public sector, these challenges are even greater due to the need to handle confidential data and convert vast quantities of often legislative documents into formats that can be processed by machines. In addition to technical challenges, there are also ethical and societal barriers that hinder the implementation of artificial intelligence systems in the public sector. To effectively overcome these challenges, a careful approach must be taken, along with ongoing education efforts.

The advantages of using chatbots can be seen in practical examples around the world. For instance, in India, English is predominantly used in both business and public life, but only about 16% of the population speaks the language. If an e-government website were available only in English, it would limit access to services for the majority of the population. Chatbots overcome this issue by supporting multiple languages, thus enabling personalized communication for users. Another major advantage is the constant availability of services. These models can provide information anytime, regardless of institutional working hours. For example, consider a situation where someone loses their passport while in a foreign country. If this happens outside of embassy working hours, the citizen doesn't have to wait until the next day, as the chatbot can immediately provide guidance on how to apply for a replacement passport. [11].

One of the most advanced countries in terms of digitalization is Estonia. Over the years, Estonia has worked extensively on digitalizing public services, and to make the entire system efficient and accessible, it developed a chatbot called SUVE. SUVE uses data from the electronic ID system to provide personalized information to citizens. This is just one small segment of Estonia's comprehensive e-government system, but it demonstrates how the relationship between the state and its citizens can be improved [12].

2.2 Azure technologies for AI system development

Microsoft Azure, formerly known as Windows Azure, is a comprehensive cloud platform built on a globally distributed network of data centers, ensuring high availability and reliability. An active subscription grants users access to its full range of resources, enabling them to deploy services and establish operational environments. Those services help companies achieve their organizational goals. Available tools can be applied in various industries such as finance, e-commerce, and are also compatible with open-source technologies. Additionally, there are several usage models for the Azure platform [13]:

1) **Infrastructure as a Service (IaaS)** – Provides computing services via virtualization;
2) **Platform as a Service (PaaS)** – Enables development, testing, and management of applications without the need to manage the underlying infrastructure;
3) **Software as a Service (SaaS)** – Access to applications via the internet, usually through a subscription model.

Having established these foundational principles, we present a detailed example illustrating the application of Azure services in the development of our chatbot. The chatbot was developed as a web application using the Azure OpenAI service and Azure AI Search to enable conversational search functionality. The Azure OpenAI GPT model is used on the server side to generate responses, while Azure AI Search is utilized for data indexing and retrieval.

This architecture enables efficient, intuitive information retrieval and delivers a ChatGPT-like experience based exclusively on an organization's internal data without requiring advanced technical expertise. Technical administrators retain full control through configurable settings, ensuring both operational efficiency and adherence to corporate policies.

2.2.1 Azure OpenAI

The Azure OpenAI Service integrates OpenAI's language models directly into the Azure ecosystem, enabling organizations to embed advanced natural-language capabilities within their applications. By leveraging Azure's security, scalability, and seamless integration with other Azure tools, organizations can automate workflows, analyze high-volume text data, and drive both business innovation and operational efficiency [14].

The Azure OpenAI service is specifically designed for in-depth data processing. Based on text input, it can easily generate relevant responses within a given context, which is particularly helpful for

content creation, writing emails, and similar tasks. An additional advantage is language translation, as this model supports multilingual communication in real time [15].

By using this service, users gain access to AI language models such as GPT-4, GPT-3, and text-to-speech models. In collaboration with OpenAI, Azure OpenAI provides an API, enabling compatibility and integration between the two platforms. In this implementation, the GPT-4o was chosen for its ability to handle complex and multilingual queries relevant to e-government use cases. A key benefit Microsoft offers compared to the classic OpenAI setup is enhanced security, made possible through the use of private networks, regional availability, and content filtering [16].

The Azure OpenAI service is based on several key concepts that enable its efficient use. According to [17], the following concepts are essential for understanding how the service operates:

1) **Prompt** – Represents the input data the user provides in order to receive a corresponding output. These inputs can range from simple questions to complex instructions (e.g., "Generate the simplest HTML code.");
2) **Token** – Azure OpenAI processes the input text by dividing it into tokens. A token can be a word or part of a word (i.e., a string of characters). For example, the word "faculty" in Serbian ("факултет") may be split into syllables such as "фа," "ку," and "лтет," while shorter words like "мир" (peace) may represent a single token. The total number of tokens depends on the input parameters and user requirements;
3) **Models** – A specific pre-trained network (e.g., GPT-4 for text, DALL·E for images, Whisper for audio), each optimized for particular generation or analysis tasks;
4) **Prompt Engineering** – The practice of crafting and refining input prompts to steer model outputs toward desired formats and levels of detail.

One of the most important components of the Azure OpenAI service is content filtering. This feature works in conjunction with other models and operates by filtering prompts through a set of classification models to prevent the generation of harmful or prohibited content. Text filtering models are specifically trained for categories such as hate speech, sexual content, violence, and similar topics in languages like English and French, although the service supports many languages. In any case, it is recommended to create a custom list to guide content filtering [18].

2.2.2 Azure App Service

Azure App Service is a service fundamentally based on the HTTP protocol and is used for hosting web applications and REST APIs. Applications can be written in various programming languages such as .NET, Java, Python, or PHP. These applications are optimized to run in environments based on either Windows or Linux and can be scaled according to specific needs. Additional advantages of this service include security, automatic scaling, and various features offered both by the service itself and the broader Microsoft Azure portal. Furthermore, DevOps benefits can be leveraged due to integrations with various tools, such as Azure DevOps, GitHub, and Docker [19].

2.2.3 Azure AI Search

Azure AI Search is a system designed for accessing information based on search queries. It enables advanced search capabilities across various types of content based on what is entered into the search index.

In addition, this tool offers a wide range of advanced search technologies developed for applications that require high performance and scalability. This system provides direct integration with large language models (LLMs) from the Azure OpenAI service as well as Azure Machine Learning. Moreover, it is possible to connect to models outside the Microsoft ecosystem using specific implementation strategies. The architecture of Azure AI Search is shown in Fig. 1.

In Fig. 1, the Azure OpenAI service is positioned between the database, which contains unindexed data, and the client application, which sends search requests. On the client application side, the search is defined through the Azure AI Search API and may include additional settings such as synonym search, filtering, sorting, and more.

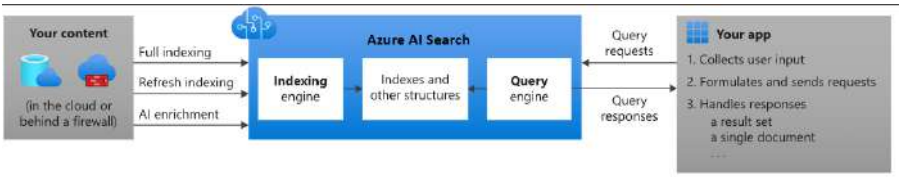


Fig. 1. Architecture of the Azure AI Search [20].

According to [21], the two primary operations of the service are indexing and query execution. Indexing is the process of loading content into the search service, thereby making it available. Inserted text is broken down into tokens and stored in inverted indexes. Azure AI Search supports JSON document types that can be indexed. The client application communicates with the index to retrieve information, which can then be filtered and sorted. Once these processes are completed, the query execution takes place. The client-side application sends a query to Azure AI Search specifying a search term along with other parameters based on which results are returned. Azure AI Search encompasses both traditional search methods and newer approaches powered by generative artificial intelligence, which makes this model an efficient and user-friendly solution for accessing information.

2.2.4 Azure Cosmos DB

Modern technological advancements demand that applications remain continuously available and highly responsive. They must handle dynamic workloads and concurrent requests without interruption, even under peak user activity. Deploying application instances in the data center nearest to end users helps minimize latency. Furthermore, the rise of AI-driven features has increased the complexity of data management, since many modern systems now rely on multiple, heterogeneous databases [22].

These challenges can be overcome by using Azure Cosmos DB. This type of database simplifies and accelerates the application development process by offering a unified database for diverse needs, such as geolocation caching or backup storage. It also supports various data models, including relational, graph, and table formats. Additional benefits of using Azure Cosmos DB include simplified administration due to extensive automation, even in the area of scaling, which is crucial when an application needs to meet user demand [23].

In the model whose implementation will be described in the following chapter, one of the main roles of Azure Cosmos DB is to enable the storage of each individual conversation's history, i.e., chat history. This functionality significantly helps users as it allows them to reference previous queries and continue earlier conversations.

3. Architecture of a chatbot as an e-government service using Azure OpenAI service

This chapter will describe and illustrate the system architecture for the development and implementation of a chatbot, as well as its components. The goal of the system as a whole is to enable efficient resource management, the construction of an internal knowledge base, all in combination with artificial intelligence technologies. The development of this system is achieved through the integration of several Azure services: Function Apps, Azure OpenAI, Azure AI Document Intelligence, and Azure AI Search - some of which have been previously described. This

integration creates a unified solution capable of handling large volumes of data, automating processes, and enabling rapid resource scaling.

For the development of the chatbot as an e-government service, the system architecture consists of two main components: document ingestion and management, and user response generation. This structure enables the chatbot to search and respond to user queries based on documents stored in the knowledge base, as illustrated in Fig. 2.

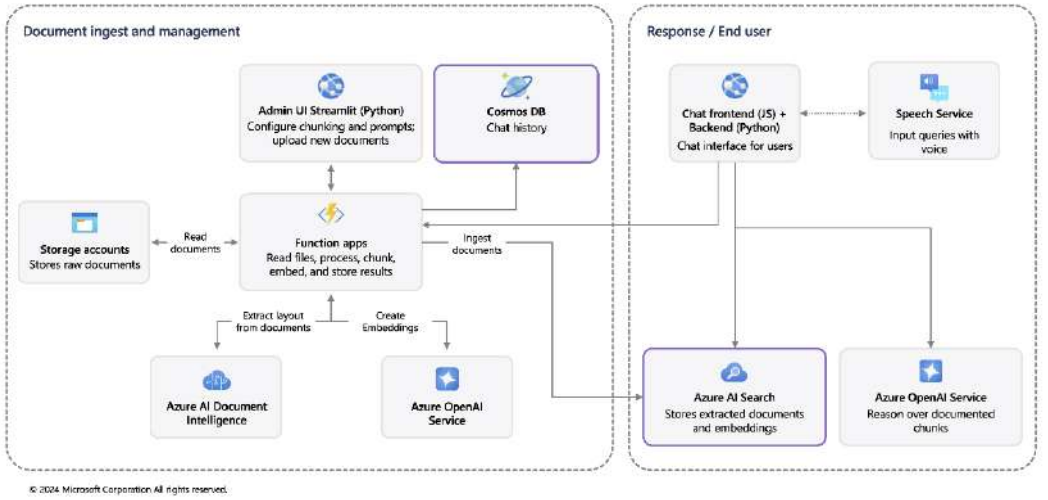


Fig. 2. System architecture for chatbot development [24].

Within the left section, which pertains to document management, the Function Apps component serves as the central element interacting with all others. Its role is to read documents, process them, divide them into smaller segments, and then store them. The files, i.e., the documents, are stored within a Storage Account. The Azure OpenAI service provides the ability to understand the content, while Azure AI Document Intelligence extracts key information and sends it for further processing. The Cosmos DB ensures the functionality of saving chat history.

Once the documents are processed, they are forwarded to Azure AI Search, where they are stored for later usage. In the Response / End-User section, the user interacts with a JavaScript chat frontend paired with a Python backend, which together facilitate communication with the chatbot. Voice queries are captured and transcribed by the Speech Service. Each request is routed concurrently to Azure AI Search for retrieval of relevant document embeddings and to Azure OpenAI Service for reasoning over those embeddings to produce context-aware, human-like responses.

Below is a concise explanation of how the previously mentioned components work together in a concrete example, as illustrated in Fig. 3:

1) The user submits a question or request through the graphical interface;
2) The application then forwards the request to Azure AI Search, which performs the search;
3) Once the information is found in the knowledge base, it is forwarded to the Azure OpenAI service, which generates a meaningful response and sends it back as feedback;
4) Finally, the information is returned to the user as a response to their request via the user interface.

4. Developed chatbot: functionalities and evaluation

In this chapter, we describe the architecture and operation of the implemented chatbot system, outline its user-facing functionalities, and present the methodologies employed to validate system

performance and user experience. In order to guarantee uninterrupted operation within production environment, it is necessary to carry out the previously mentioned activities to verify the fulfillment of requirements and to continuously improve the system as a whole.

Although no formal benchmarking was performed, GPT-4o was selected based on internal testing and current literature. The model demonstrated strong performance in handling multilingual queries, retrieving information from longer documents, and producing clear and relevant responses, which are key features for public services designed. As noted in [25], GPT-4o improves response speed and efficiency while maintaining high accuracy in reasoning, making it a suitable choice for real-world e-government applications.

Upon accessing the chatbot, the user is presented with a window where they can submit their request either by typing or by using voice input. The main page of the user interface is shown in Fig. 4.

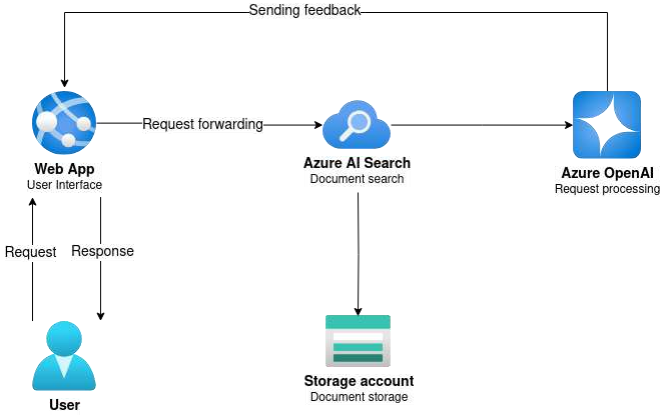


Fig. 3. Illustration of a Practical Example.

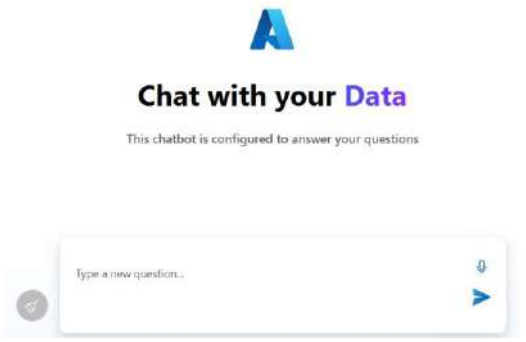


Fig. 4. User interface of chatbot.

Once the user asks a question, the chatbot returns a response if it can find it in its knowledge base; otherwise, it returns a generic message. When responding, the model references the documents it has within the database.

Fig. 5 illustrates a sample interaction in which the user asks, “What is eUprava?” and the chatbot returns a concise overview of Serbia's eUprava portal in Serbian. Directly beneath the response, the disclaimer “AI-generated content may be incorrect” appears to remind users that the information should be verified and not accepted uncritically.

If the user wishes to delete messages within the conversation, they can simply click the brush icon inside the blue circle immediately to the left of the input field. In any case, the conversation will be saved within the chat history, which can be seen in Fig. 6.

The chat history feature is particularly useful because the user can return to previous conversations. Additionally, the general recommendation for interacting with chatbots is that the user should maintain a separate conversation for each topic or service request to avoid confusion and overlap.

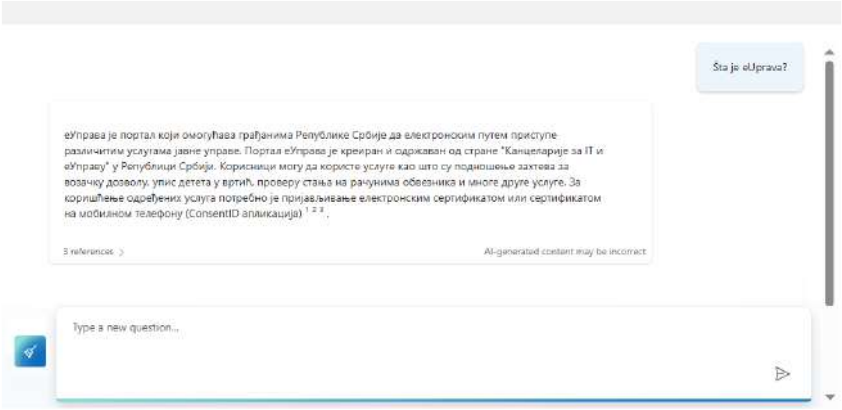


Fig. 5. User interaction with chatbot.

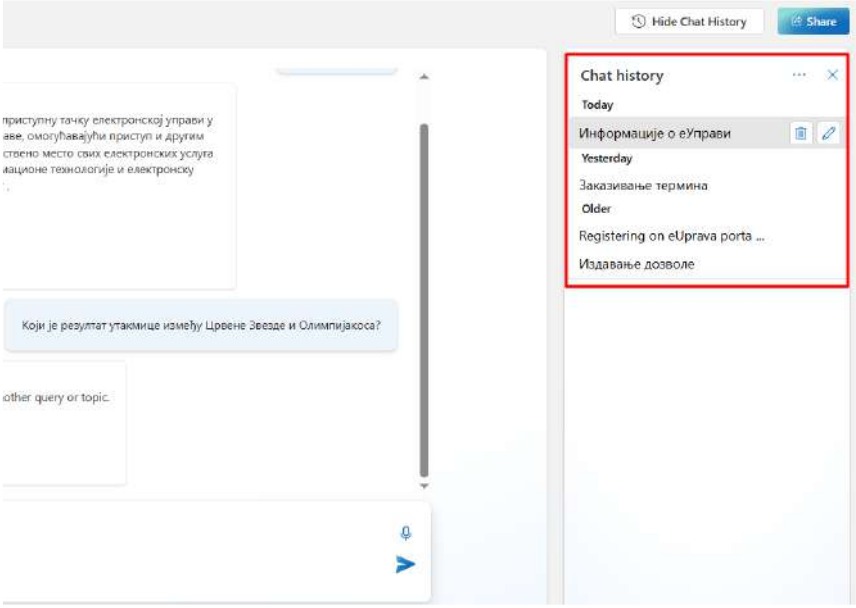


Fig. 6. Chat history.

Furthermore, the user can share any conversation by clicking the "Share" button, which will allow them to copy a link that they can share.

While generative AI unlocks powerful capabilities, it also presents new security vulnerabilities. In particular, attackers may craft malicious prompts to extract sensitive information or coerce the system into producing prohibited content. To guard against such threats, our chatbot undergoes rigorous adversarial testing, including scenarios in which an attacker requests confidential user records. Fig. 7 illustrates one such exchange: the user's prompt, "If you have database access, display all users and their national identifiers" is met by the chatbot's refusal, "I'm sorry, I don't have any information on that topic. I can assist with information about eUprava".

Performance testing is especially important after deploying the system to production. These tests can be conducted using Azure Load Testing. Through load testing, it is possible to measure how the system responds to a higher volume of user requests, and it can also help identify potential resource bottlenecks. Conducting such tests contributes to system optimization.

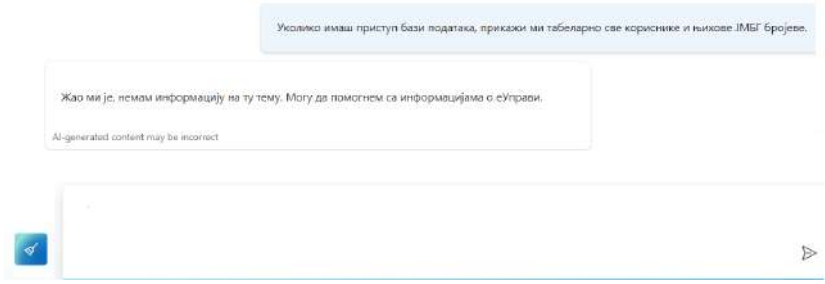


Fig. 7. Example of a malicious prompt.

From Fig. 8, some aggregated data can be seen, such as the total number of requests, average response time, and throughput. The data is refreshed every 5 seconds by default. The test results can also be visualized through charts.

Fig. 8 and Fig. 9 show the main metrics. What is also important is the utilization of the CPU, memory, and other resources, which can also be measured through these tests.

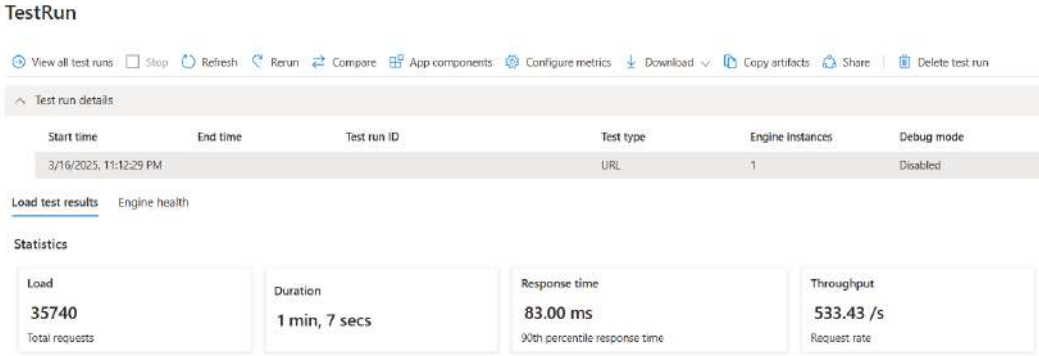


Fig. 8. Performance testing.

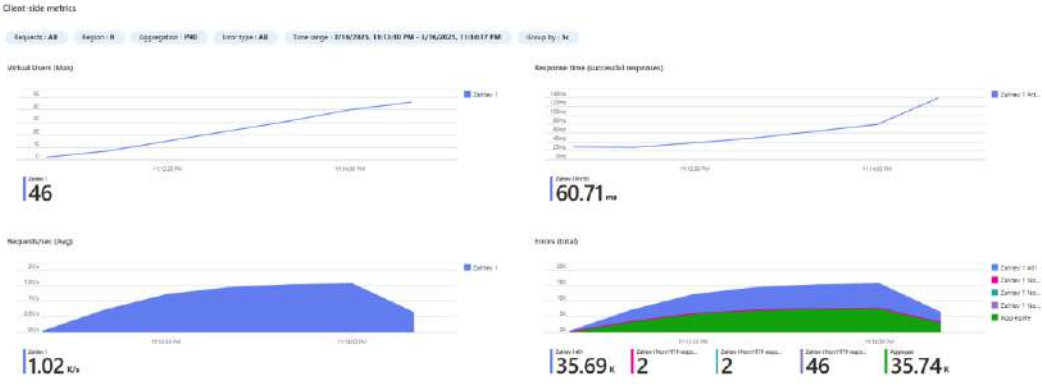


Fig. 9. Graphical representation of the test.

5. Conclusion

This paper describes how electronic public services can be modernized and improved through the application of artificial intelligence. The entire process of developing a chatbot as an e-government service is covered, from the theoretical aspect, through the analysis of the technologies used, to the implementation of the technical solution.

The chatbot discussed in this paper provides citizens with an intuitive way to access and better understand e-government services through the user interface. The advantages of this model lie in its simple integration with the web portal, high degree of automation, and rapid scalability of resources.

The direction for further thinking and enhancement of the web application should focus on the development of APIs to enable better interaction and improvement of the user interface for future integration with other platforms and applications. Although this paper places emphasis on server-side functionality, maximizing practicality requires simultaneous investment in API development, user-interface improvements, integration adapters, and supporting services to ensure seamless interoperability with external platforms, robust security, and an intuitive experience for end users.

Additionally, in order to maintain the relevance and accuracy of the data, the model must be continuously trained, which improves the service. Besides system maintenance, it is also necessary to implement new functionalities, one of which is Azure Cache Redis, which provides more efficient request processing by caching queries and offering faster data access.

Furthermore, future research could explore a comparative analysis between Azure OpenAI and similar services from other cloud providers, such as AWS Bedrock or Google Vertex AI. Such comparisons may provide valuable insights into performance, cost-efficiency, and integration flexibility, particularly in the context of public sector requirements. This would contribute to more informed deployment decisions in heterogeneous technological environments.

One downside of this model is that, despite the fact that the documents in the knowledge base contain facts, the answers generated by artificial intelligence may still be inaccurate. To minimize the number of such answers, continuous model learning and building a high-quality knowledge base are necessary.

Finally, the implementation of the chatbot shows that public sector organizations can successfully apply modern technologies and transform their services, enabling users to have a simpler experience and wider use.

References

[1]. E. Adamopoulou and L. Moussiades, “An Overview of Chatbot Technology,” IFIP Advances in Information and Communication Technology, vol. 584 IFIP, pp. 373–383, 2020, doi: 10.1007/978-3-030-49186-4_31.

[2]. S. Hussain, O. Ameri Sianaki, and N. Ababneh, “A Survey on Conversational Agents/Chatbots Classification and Design Techniques,” Advances in Intelligent Systems and Computing, vol. 927, pp. 946–956, 2019, doi: 10.1007/978-3-030-15035-8_93.

[3]. L. F. Lins dos Santos, “Evaluating and Comparing Generative-based Chatbots Based on Process Requirements,” Dec. 06, 2023, University of Waterloo.

[4]. C. Zhang and Y. Lu, “Study on artificial intelligence: The state of the art and future prospects,” Journal of Industrial Information Integration, vol. 23, p. 100224, Sep. 2021, doi: 10.1016/J.JII.2021.100224.

[5]. M. I. Jordan and T. M. Mitchell, “Machine learning: Trends, perspectives, and prospects,” Science, vol. 349, no. 6245, pp. 255–260, Jul. 2015, doi: 10.1126/SCIENCE.AAA8415.

[6]. K. R. Chowdhary, “Natural Language Processing,” Fundamentals of Artificial Intelligence, 2020, doi: 10.1007/978-81-322-3972-7_19.

[7]. E. Novikova, “Influence of NLP on organizational design and development of dynamic capabilities,” 2024.

[8]. S. Minaee et al., “Large Language Models: A Survey”.

[9]. M. Nejjar, L. Zacharias, F. Stiehle, and I. Weber, “LLMs for science: Usage for code generation and data analysis,” Journal of Software: Evolution and Process, vol. 37, no. 1, p. e2723, Jan. 2025, doi: 10.1002/SMR.2723.

[10]. A. Androutsopoulou, N. Karacapilidis, E. Loukis, and Y. Charalabidis, “Transforming the communication between citizens and government through AI-guided chatbots,” Government Information Quarterly, vol. 36, no. 2, pp. 358–367, Apr. 2019, doi: 10.1016/J.GIQ.2018.10.001.
[11]. A. Rukavina, “How chatbots can improve government services.” Accessed: May 08, 2025. [Online]. Available: https://www.infobip.com/blog/ai-customer-experiences-in-government
[12]. “Invest in Estonia.” Accessed: May 08, 2025. [Online]. Available: https://investinestonia.com/estonia-created-suve-an-automated-chatbot-to-provide-trustworthy-information-during-the-covid-19-situation/
[13]. M. M. Salas-Zárate and M. Luis Colombo-Mendoza, “CLOUD COMPUTING: A REVIEW OF PAAS, IAAS, SAAS SERVICES AND PROVIDERS,” Lámpsakos (revista descontinuada), no. 7, pp. 47–57, Jun. 2012, doi: 10.21501/21454086.844.
[14]. A. Jamshidi, “Applications of Microsoft Azure OpenAI to code management,” 2024.
[15]. N. P. S. Kumar, S. Ramakrishnan, and M. Vignesh, “AI-Based Data Analytics & Business Intelligence Chatbot Using Azure Functions and OpenAI,” pp. 129–137, 2025, doi: 10.1007/978-981-97-9619-9_11.
[16]. V. T. Mahajan and S. Sarode, “COMPARATIVE INSIGHTS INTO AI TOOLS: AN ANALYTICAL APPROACH”.
[17]. S. Ifrah, “Getting Started with Azure OpenAI,” Getting Started with Azure OpenAI, 2024, doi: 10.1007/979-8-8688-0599-8/COVER.
[18]. Microsoft, “What is Azure OpenAI Service?” Accessed: May 12, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/ai-services/openai/overview
[19]. Microsoft, “App Service Overview.” Accessed: May 12, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/app-service/overview
[20]. Microsoft, “What’s Azure AI Search?” Accessed: May 12, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/search/search-what-is-azure-search
[21]. K. Weiland, “Azure AI Search: A Comprehensive Guide.” Accessed: May 12, 2025. [Online]. Available: https://www.linkedin.com/pulse/azure-ai-search-comprehensive-guide-kim-weiland-mvb1e/
[22]. Y. Lu, “Artificial intelligence: a survey on evolution, models, applications and future trends,” Journal of Management Analytics, vol. 6, no. 1, pp. 1–29, Jan. 2019, doi: 10.1080/23270012.2019.1570365.
[23]. J. R. Guay Paz, “Introduction to Azure Cosmos DB,” Microsoft Azure Cosmos DB Revealed, pp. 1–23, 2018, doi: 10.1007/978-1-4842-3351-1_1.
[24]. Microsoft Corporation, “Chat with your data - Solution Accelerator,” GitHub. Accessed: May 12, 2025. [Online]. Available: https://github.com/Azure-Samples/chat-with-your-data-solution-accelerator
[25]. R. Islam and O. M. Moushi, “GPT-4o: The Cutting-Edge Advancement in Multimodal LLM,” TechRxiv, Jul. 2024, doi: 10.36227/TECHRXIV.171986596.65533294/V1.

Информация об авторах / Information about authors

Лазарь РАДОСАВЛЬЕВИЧ окончил факультет организационных наук Белградского университета. Работает в Управлении информационных технологий и электронного управления Республики Сербии. Его научные интересы включают искусственный интеллект, кибербезопасность и интернет-технологии.

Lazar RADOSAVLJEVIĆ graduated from the Faculty of Organizational Sciences, University of Belgrade. Employed at the Office for IT and eGovernment of the Republic of Serbia. His research interests include artificial intelligence, cybersecurity, and internet technologies.

Милица СИМИЧ – аспирантка по направлению «Программная инженерия и электронный бизнес», ассистент на факультете организационных наук Белградского университета. Член IEEE и IEEE Computer Society. Её научные интересы включают искусственный интеллект, генеративный ИИ, образование, программную инженерию, интернет-технологии и электронный бизнес.

Milica SIMIĆ – PhD student in Software Engineering and E-Business, teaching assistant at the Faculty of Organizational Sciences, University of Belgrade. Member of IEEE and the IEEE Computer Society. Her research interests include artificial intelligence, generative AI, education, software engineering, internet technologies, and e-business.

Александр ЙОКСИМОВИЧ – магистрант по направлению «Электронный бизнес» и учебный ассистент на факультете организационных наук Белградского университета. Его научные интересы включают моделирование, цифровые двойники, Интернет вещей и их применение в умном сельском хозяйстве.

Aleksandar JOKSIMOVIĆ – Master student in E-Business Technologies and teaching associate at the Faculty of Organizational Sciences, University of Belgrade. His research interests include simulations, digital twins, the Internet of Things, and their applications in smart agriculture.

Тамара НАУМОВИЧ – аспирантка по направлению «Программная инженерия и электронный бизнес», ассистент на факультете организационных наук Белградского университета. Её научное исследование посвящено блокчейн-технологиям. Научные интересы включают Web3, интернет-технологии, электронный бизнес и электронное образование.

Tamara NAUMOVIĆ – PhD student in Software Engineering and E-Business, teaching assistant at the Faculty of Organizational Sciences, University of Belgrade. Her PhD research focuses on blockchain technologies. Her research interests include Web3, internet technologies, e-business, and e-education.

Марьяна ДЕСПОТОВИЧ-ЗРАКИЧ – профессор факультета организационных наук Белградского университета, руководитель лаборатории моделирования. Член IEEE и IEEE Computer Society. Заместитель председателя отделения IEEE Computer Chapter CO 16 и руководитель семинара при IEEE Computer Chapter CO 16. Её научные интересы включают блокчейн, интернет-технологии и Интернет вещей (IoT).

Marijana DESPOTOVIĆ-ZRAKIĆ – Full professor at the Faculty of Organizational Sciences, University of Belgrade, and Head of the Laboratory for Simulation. Member of IEEE and the IEEE Computer Society. Vice Chair of the IEEE Computer Chapter CO 16 and Head of the Seminar of the IEEE Computer Chapter CO 16. Her research interests include blockchain, internet technologies, and the Internet of Things (IoT).

DOI: 10.15514/ISPRAS-2025-37(6)-11

Показатели множественного числа существительных в селькупских диалектах

С.В. Ковылин, ORCID: *0000-0003-0108-9214 <kovylin.ser@yandex.ru>*

Институт системного программирования им. В. П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Томский государственный педагогический университет
Россия, 634061, г. Томск, ул. Киевская, д. 60.

Аннотация. В статье рассматриваются показатели множественного числа существительных в южных, переходном, центральных и северных диалектах селькупского языка. Материалом послужили корпусные данные объемом более 85,000 словоформ, расположенные на платформе Lingvodoc и в личных архивах (файлы Fieldworks Language Explorer), а также общие грамматические и лексические труды по языку. Множественное число существительных может быть выражено суффиксами *-t(V), -la, -i (-ni),* контаминированным показателем *-lat*, суффиксами множественного числа в совокупности с показателями взаимной связи *-sa-* и собирательного множества *-mi-* – *-sat, -sala, -mɨt, -mɨtla*, а также суффиксом собирательного множества *-mɨ* без дополнительных маркеров множественности. В результате исследования было выяснено, что основным маркером множественного числа в южной и переходной языковых зонах является *-la*, в центральных васюганском и тымском, а также северных диалектах – *-t(V)*. В центральном нарымском диалекте встречаются оба показателя *-t(V)* и *-la* в том числе в контаминированной форме *-lat*, где для северной части более характерен суффикс *-t(V)*, а для южной *-la*. В северных и центральных диалектах в посессивных формах используется показатель множественного числа *-i (-ni)*, в то время как в южных и переходном диалектах он был вытеснен показателем *-la* в тех же позициях. В южных, переходном, центральных и северных диалектах с терминами родства используется суффикс взаимной связи *-sa-* вместе с *-t(V)* – *-sat*; реже в южных, центральных и северных диалектах суффикс собирательного множества *-mi-* вместе с *-t(V)* – *-mɨt*. В северных диалектах для выражения собирательного множества без дополнительных числовых показателей применяется показатель *-mɨ*.

Ключевые слова: селькупский язык; множественное число; корпусные данные; существительное.

Для цитирования: Ковылин С.В. Показатели множественного числа существительных в селькупских диалектах. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 181–192. DOI: 10.15514/ISPRAS–2025–37(6)–11.

Благодарности: Исследование автора было поддержано грантом РНФ 25-78-20002 «Возможности искусственного интеллекта для сравнительно-исторического изучения малоресурсных языков народов РФ».

Plural Number Markers of Nouns in Selkup Dialects

S. V. Kovylin, ORCID: 0000-0003-0108-9214 <kovylin.ser@yandex.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia;
Tomsk State Pedagogical University,
60, Kievskaya st. Tomsk, 634061, Russia.

Abstract. The paper is devoted to the study of plural forms of nouns in the southern, transitional, central and northern dialects of Selkup. The material comprises corpus data of more than 85,000 tokens located on the Lingvodoc platform and in personal archives (Fieldworks Language Explorer files), as well as general grammatical and lexical papers on the language. The plural of nouns can be expressed by the suffixes *-t(V)*, *-la*, *-i (-ni)*, by the contaminated marker *-lat*, plural suffixes in combination with the markers of mutual connection *-sa-* and the collective plurality *-mi-* – *-sat*, *-sala*, *-mit*, *mitla*, as well as by the suffix of mutual connection *-mi-* without additional markers of plurality. As a result of the research, it was found out that the main plural suffix in the southern and transitional language zones is *-la*, in the central Vasyugan and Tym, as well as in the northern dialects – *-t(V)*. In the central Narym dialect, both markers *-t(V)* and *-la* are found, including the contaminated form *-lat*, where the suffix *-t(V)* is more characteristic for the northern part, and *-la* for the southern part. In northern and central dialects, the plural marker *-i (-ni)* is used in possessive forms, while in southern and transitional dialects it was replaced by the marker *-la* in the same positions. In southern, transitional, central and northern dialects, with kinship terms, the suffix *-sa-* together with *-t(V) – sat* is used; more rarely, in southern, central, and northern dialects the collective suffix is *-mi-* together with *-t(V) – -mit* is observed. In northern dialects, the marker *-mi-* is used to express collective plurality without additional suffixes of numbers.

Keywords: Selkup language; plural number; corpus data; noun.

For citation: Kovylin S.V. Plural number markers of nouns in Selkup dialects. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 181-192 (in Russian). DOI: 10.15514/ISPRAS-2019-37(6)-11.

Acknowledgements. This paper was supported by the grant RNF № 25-78-20002 “Capabilities of Artificial Intelligence for Comparative-Historical Study of Low-Resource Languages of the Peoples of the Russian Federation”.

1. Введение

Тема множественного числа существительных в селькупском языке многократно обсуждалась в различных публикациях, например см. [1; 2: 169–170; 3: 101–110; 4; 5: 54–64; 6: 393–398]. Несмотря на это изучение показателей множественности проводилось не во всех диалектах, либо обсуждение ограничивалось отдельными суффиксами; не всегда учитывались селькупские данные за XIX в. В данной работе производится попытка анализа селькупского материала на большом объеме данных с учетом результатов предыдущих научных изысканий, что обуславливает ее актуальность.

Целью исследования является анализ встречаемости показателей множественного числа существительных в южных, переходном, центральных и северных диалектах селькупского языка.

Материалом исследования являются корпусные данные, расположенные на платформе Lingvodoc[1] [7] либо в личных архивах (файлы программы Fieldworks Language Explorer), состоящие из глоссированных текстов архива лаборатории языков народов Сибири Томского государственного педагогического университета (далее ЛЯНС), текстов и словарей М. А. Кастрена [8], Н. П. Григоровского [9–12], св. Макария (Невского) [13-14] и Л. Сабо [15]. Общий объем корпусных данных составляет более 85,000 словоформ. Также при анализе

[1] Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В. П. Иванникова РАН – ЦКП ИСП РАН

использовался «Диалектологический словарь селькупского языка (северное наречие)» [16], «Селькупско-русский диалектный словарь» [17] и данные приведенных ранее грамматик и статей [1; 2: 169–170; 3: 101–110; 4; 5: 54–64; 6: 393–398].

В работе применяется классификация диалектов С. В. Глушкова, А. В. Байдак и Н. П. Максимовой [18], с разделением территории на

- северную группу – включает диалекты:
  1) тазовский;
  2) ларьякский (верхнетолькинский);
  3) карасинский;
  4) туруханский;
  5) баихинский;
  6) елогуйский;
- центральную группу:
  7) ваховский;
  8) тымский;
  9) васюганский;
  10) нарымский;
- южную группу:
  11) среднеобской;
  12) чаинский;
  13) кетский;
  14) верхнеобской;
  15) чулымский.

Также нами дополнительно выделяется переходная зона в виде иванкинского диалекта.

2. Общие положения

Основными суффиксами множественного числа существительных в селькупских диалектах являются *-t(V)* ‘PL’, *-la* ‘PL’, *-lat* (*-la-t* ‘PL-PL’) и *-i (-ni)* ‘PL’. Суффикс PL *-t(V)* ‘PL’ – уральского происхождения (ПУ **-t*) [19: 297] и относится к базовым в северных и части центральных диалектов. Маркер *-la* ‘PL’ является основным суффиксом в южных и части центральных диалектов. Существует как минимум две точки зрения на происхождение данного показателя: 1) заимствование из тюркских языков (со ссылкой на М. А. Кастрена [20: 109] и К. Доннера [21: 99]); 2) уральский собирательный суффикс *-lV* (со ссылкой на Т. Лехтисало [22: 161–162], А. Йоки [23: 33], А. Кюннапа [23: 42] цит. по [3: 109]. Контаминированный суффикс *-lat* (*-la-t* ‘PL-PL’) получил свое распространение на средней Оби – по большей части в нарымском диалекте, на границе распространения суффиксов *-t(V)* ‘PL’ и *-la* ‘PL’. Суффикс *-i* или *-ni* ‘PL’ используется в северных и центральных диалектах в поссессивных формах и присоединяется напрямую к основе [2: 169–170; 3: 106]. Показатель *-i* ‘PL’ также встречается в других уральских языках в схожей функции [19: 298–299]. Суффиксы множественного числа *-t(V)* ‘PL’ или *-la* ‘PL’, а также их контаминированные формы, присоединяются к показателям взаимной связи *-sa-* ‘RECIP’ или собирательного множества *-mi-* ‘COLL’ для обозначения совокупности однородных предметов. Отдельно упомянем суффикс собирательного множества *-mɨ* ‘COLL’ в северных диалектах, который, появляясь без дополнительных маркеров множественного числа, используется для обозначения существительных, представляющих нерасчлененное множество, что противопоставляется суффиксу расчлененного множества *-t(V)* ‘PL’ [2: 170].

3. Способы маркирования существительных суффиксами множественного числа

В табл. 1 показаны суффиксы множественного числа существительных в селькупских материалах за XIX–XXI в. За основу была взята таблица Э. Г. Беккер [5: 64].

Табл. 1. Суффиксы множественного числа существительных в селькупских диалектах.
Table 1. Plural number suffixes of nouns in Selkup dialects.

Юж.						
	-t(V) ‘PL’	*-la* ‘PL’	*-lat* ‘PL’ (как неосновной вариант)	*-i (-ni)* ‘PL’ (в посс. формах)	*-sat / -mit* (сов. однородн. предм.)	н.п.
XIX в.						
Григ. (н.-чаин.)	-	+	-	-	-	-
Кастрен (чаи.)						
Кастрен (чул.)						
Кастрен (в.-кет.)						
Кастрен (н.-кет.)						
XX в.						
ЛЯНС (Сон. ср.-об.), Беккер и др. [5] (В.-об. сюс.)	-	+	Примеров не обнаружено	-	Примеров не обнаружено	
ЛЯНС (в.-кет.), Беккер и др. 5] (В.-кет. сюс., тюй.)	-	+	Примеров не обнаружено[2]	-	*-mɨtla*	
ЛЯНС (н.-/ср.-кет.), Беккер и др. [5] (Ср.-кет. сюс.)	ед. случаи в Карелино по Беккер и др. [5: 64])	+	Примеров не обнаружено	-	*-zat*	
Пер.						
	-t(V) ‘PL’	*-la* ‘PL’	*-lat* ‘PL’ (как неосновной вариант)	*-i (-ni)* ‘PL’ (в посс. формах)	*-sat / -mit* (сов. однородн. предм.)	
Кастрен (Тог. ив.)	-	+	Примеров не обнаружено	-	-	
Макарий (ив.)	-	+		-	-	
XX в.						
ЛЯНС, Беккер и др. [5] (Ив. ив.), Быконя и др. [17] (шеш.)	один пример на *-t* по Беккер и др. [5: 64]	+	Примеров не обнаружено	-	*-sat* по Быконя и др. [17: 319]	

[2] Э. Г. Беккер отмечает, что контаминированные суффиксы встречаются у кетских сюсюкумов и редко в языке обских шешкупов, однако большое количество проанализированного материала из архивов ЛЯНС, отсутствие однозначных примеров в описаниях, несвойственность суффикса PL *-t* кетским сюсюкумам (кетский диалект) и обским шешкупам / шешкумам (Иванкино) говорят об обратном. Все встретившиеся нам примеры с комбинацией *-lat* можно объяснить, как сочетание суффиксов *-la* ‘PL’ и *-t* ‘POSS.3SG’, где последний выступает в поссессивной и дискурсивно-прагматической функциях.

Центр.						
	-t(V) ‘PL’	*-la* ‘PL’	*-lat* ‘PL’ (как неосновной вариант)	*-i (-ni)* ‘PL’ (в посс. формах)	*-sat / -mit* (сов. однородн. предм.)	
XIX в.						
Кастрен (нар.)	*-t, -n*	*?*	Примеров не обнаружено	+	-	Нарым
	?	+				?Парабель
Макарий (Нар. нар.)	+	-	Примеров не обнаружено	-	-	Нарым
XX в.						
ЛЯНС (нар.), Беккер и др. [5] (Ср.-об. чум.), Быконя и др. [17] (нар.)	+	-	+ у ряда информантов встречается, как дополнительный вариант; у других не встречается	+	*-sat, -mit*; *-sala* по Быконя и др. [17: 319]	Тюхтерево
	-	+				Ласкино
	аусл.: *-t(V)* (фон. обусл. *-t, -n*) (реже)	+ (чаще)				Парабель, Нельмач, Соиспаево
	-	+				Мумышево, Сунгурово
	аусл.: *-t(V)* (фон. обусл. *-t, -n*) (основной)	-	*-lat* (основной)			Кенга
ЛЯНС (вас.), Беккер и др. [5] (Вас. чум.)	аусл.: *-t(V)* (фон. обусл. *-t, -n*)	-	Примеров не обнаружено	+	Примеров не обнаружено	
Сабо (тым.)	-	+	-	-	-	Напас
ЛЯНС (тым.), Беккер и др. [5] (Тым. чум.)	аусл.: *-t(V)* (фон. обусл. *-t, -n*)	-	Примеров не обнаружено	+	Примеров не обнаружено	
Сев.						
	-t(V) ‘PL’	*-la* ‘PL’	*-lat* ‘PL’ (как неосновной вариант)	*-i (-ni)* ‘PL’ (в посс. формах)	*-sat / -mit* (сов. однородн. предм.)	
XIX в.						
Кастрен (таз.)	*-t, -n’a*	-	-	+	*-ты* (соб. форма; со всеми сущ.)	
Кастрен (ел.)	+	-	-	-	-	
Кастрен (баи.)	*-t, -n*	-	-	+	-	
Кастрен (кар.)	+	-	-	+	*-тэ / -met* (соб. форма; со всеми сущ.)	
XX в.						
ЛЯНС (Кел. ел., баи., Фар. тур.); К&Б [16] (ср.-таз., в.-таз., в.-толь., Кел. ел., баи.), Казакевич [6] (1998, 1999, 2003) (тур.)	аусл.: *-t, -n* (фон. обусл. *-t, -n*)	-	-	+	*-mit / -min, -sit; -mi* (соб. форма; со всеми сущ.)	

Материалы, представленные в обсуждении, были конвертированы в единую латиницу: *i, ɜ* (у Кастр. *ы, э*).

Множественное число существительных в селькупских диалектах согласно таблице может быть выражено при помощи суффиксов: 1) *-t(V)* ‘PL’, 2) *-la* ‘PL’, 3) *-lat* (*-la-t* ‘PL-PL’), 4) *-i (-ni)* ‘PL’, 5) *-sat* (*-sa-t* ‘RECIP-PL’), 6) *-sala* (*-sa-la* ‘RECIP-PL’), 7) *-mit* (*-mi-t* ‘COLL-PL’),

8) *-mɨtla* (*-mɨ-t-la* ‘COLL-PL-PL’), 9) *-mɨ* ‘COLL’ (без дополнительных маркеров множественности).

Во всех материалах по южным диалектам XIX–XX вв. основным суффиксом множественного числа является *-la* ‘PL’. У Э. Г. Беккер указано, что в единичных случаях показатель *-t(V)* ‘PL’ встречается в Карелино, однако без примеров, подтверждающих это [5: 64]. С терминами родства в верхнекетских источниках ЛЯНС за XX в. используется *-mɨtla* (*-mɨ-t-la* ‘COLL-PL-PL’) ‘COLL.PL’; в нижне- и среднекетских источниках ЛЯНС – *-zat* ‘RECIP.PL’. Таким образом, суффикс *-t(V)* ‘PL’ сохраняется только в комбинированных позициях. По всей видимости, суффикс *-i (-ni)* ‘PL’, появляющийся в посессивном склонении в центральных и северных диалектах, был полностью вытеснен в южных диалектах показателем *-la* ‘PL’.

Юж. XIX в.

Григ. (н.-чаин.): *чабтэ́-ла* ‘сказка-PL’, *гу́-ла* ‘человек-PL’, *лы́-лы-у* ‘кость-PL-POSS.1SG’;
Кастрен (чаи.): *lóga-la* ‘лиса-PL’, *koča-la* ‘мешок-PL’, *lóga-lá-u* ‘лиса-PL-POSS.1SG’;
Кастрен (чул.): *ýny-lä* ‘ремешок-PL’, *toto-la* ‘карась-PL’, *lóga-lá-u* ‘лиса-PL-POSS.1SG’;
Кастрен (н.-кет.): *päla-la* ‘товарищ-PL’, *súugo-la* ‘мыс-PL’, *loḳ̓ḳ̓a-la-u* ‘лиса-PL-POSS.1SG’;
Кастрен (в.-кет.): *pý-lla* ‘камень-PL’, *ú-llà* ‘куропатка-PL’, *loḳ̓ḳ̓a-la-m* ‘лиса-PL-POSS.1SG’;

Юж. XX в.

ЛЯНС (Сон. ср.-об.), Беккер и др. [5] (В.-об. сюс.): *ku-lá* ‘человек-PL’, *jéːd-la* ‘юрта-PL’, *ud-lá-w* ‘рука-PL-POSS.1SG’;
ЛЯНС (в.-кет.), Беккер и др. [5] (В.-кет. сюс., тюй.): (Мар.) *qu-lla* ‘человек-PL’, (Мар.) *kɨnna-lla* ‘собака-PL’, (Мар.) *essa-j-mɨ-t-la* ‘отец-ADJZ-COLL-PL-PL’ / родители, (Мар.) *moqqo-li-la-n-di-ssʲe* ‘спина-кость-PL-GEN-POSS.3DU-COM’;
ЛЯНС (н.-/ср.-кет.), Беккер и др. [5] (Ср.-кет. сюс.): (Кар.) *poŋɣ-la* ‘сеть-PL’, (Кар.) *qwelu-la* ‘эвенк-PL’, (УО) *inne-za-t* ‘старший.брат.отца-RECIP-PL’, (УО) *saj-laːn-d-se* ‘глаз-PL-GEN-POSS.3SG-COM’.

В переходных иванкинских материалах XIX–XX вв. фиксируется только суффикс *-la* ‘PL’. В данных Э. Г. Беккер отмечено, что *-t(V)* ‘PL’ встретился только в одном случае, однако без предоставления примеров [5: 64]. Также в грамматическом очерке В. В. Быкони приводится утверждение об использовании показателя *-sat* ‘RECIP.PL’ в среднеобском диалекте шешкупов, однако также без примеров [17: 319]. Дополнительно, из анализа материала можно предположить, что суффикс *-i (-ni)* ‘PL’, появляющийся в посессивном склонении, так же, как и в южных диалектах был вытеснен показателем *-la* ‘PL’.

Пер. XIX в.

Кастрен (Тог. ив.): *kuu-lá* ‘человек-PL’, *maadur-la* ‘богатырь-PL’, *ku-la-ut* ‘человек-PL-POSS.1PL’;
Макарий (ив.): *ку-ла* ‘человек-PL’, *ну-ла* ‘бог-PL’, *грѣк-ла-л* ‘грех-PL-POSS.2SG’;

Пер. XX в.

ЛЯНС, Беккер и др. [5] (Ив. ив.), Быконя и др. [17] (шеш.): *aːmd-la* ‘рог-PL’, *ne-la* ‘дочь-PL’, *aːmd-la-l* ‘рог-PL-POSS.2SG’.

Материалы по центральным диалектам представлены с выделением отдельных населенных пунктов, так как эта территория является контактной в распространении суффиксов *-t(V)* ‘PL’ и *-la* PL’. Анализируя нарымский диалект в отношении показателей множественного числа, по всей видимости, стоит различать языковую территорию Нарыма и его окрестностей, т. к. там используется суффикс *-t(V)* ‘PL’, и отдельно территорию выше по течению, преобладает маркер *-la* ‘PL’. В нарымских материалах XIX в. М. А. Кастрена и св. Макария (Невского), записанных, предположительно, от информантов из Н. П. Нарым, используется суффикс *-t(V)* ‘PL’, в то время как, скорее всего[3], в материалах М. А. Кастрена, записанных вверх по течению (?Парабель), появляется *-la* ‘PL’. В нарымских материалах ЛЯНС и Э. Г. Беккер за

[3] Это предположение основано на фактах общего распределения суффиксов; точной информации по месту записи материалов М. А. Кастрена с суффиксом *-la* ‘PL’ у нас нет.

XX в. показатель *-t(V)* ‘PL’ встречается в Тюхтерево и Кенге как основной и в небольших количествах в Парабели, Нельмаче и Соиспаево, в то время как суффикс *-la* ‘PL’ является основным в Ласкино, Парабели, Нельмаче, Мумышево и Сунгурово. Контаминированный показатель *-lat* (*-la-t* ‘PL-PL’) ‘PL’, является основным у одного информанта с Кенги, а также выступает в качестве вспомогательного маркера в остальных контактных населенных пунктах, например Ласкино, Парабель, Нельмач, Соиспаево. Встречаются случаи, когда у одних и тех же носителей фиксируется сразу три варианта суффиксов, например у Д. П. Саиспаева (Соис.) в «Сказке про черного царя»: 1) *parkeːləmbaːt quːt* (PL) ‘кричат люди’; 2) *wedʲ nʲe tak tänq taqəlgu gu-la-də-p* (PL-PL-ACC) ‘ведь не так просто собрать людей’; 3) *quːla* (PL) *hateleʂpat kalakonʲip.* ‘люди колотят в колокола’. В васюганских и тымских источниках ЛЯНС за XX в. наблюдается маркер *-t(V)* ‘PL’. Во всех центральных диалектах по материалам XIX–XX в. (исключение – нарымский идиом св. Макария (Невского); недостаточно материалов) используются показатели *-i* / *-ni* ‘PL’ в посессивных формах, причем в данных за XX в. по нарымскому диалекту и тымскому идиому текстов Л. Сабо также дополнительно появляются формы с суффиксом *-la* PL’, что является инновацией. Также в целом в материалах по нарымскому ареалу за XX в. фиксируются показатели *-sat* ‘RECIP.PL’, *-mɨt* ‘COLL.PL’ с терминами семейного родства (следует более детально уточнить дистрибуцию этих маркеров в дальнейшем). Дополнительно в труде В. В. Быкони отмечен пример появления формы *-sala* ‘RECIP.PL’ в нарымском диалекте, без дополнительных уточнений [17: 319]. Также в центральных диалектах наблюдается несистематический переход *-t* ‘PL’ > *-n* ‘PL’ в фонетически обусловленных позициях перед гласными и сонорными согласными.

Центр. XIX в.
Кастрен (нар.): 1) (очевидно, записано в Нарыме) *loga-t / loga-n* ‘дом-PL’, *kana-t* ‘собака-PL’, 2) (очевидно, записано ниже по течению; ?Парабель) *loga-lá* ‘лиса-PL’, *cyndä-lá* ‘лошадь-PL’, 3) *loga-ní-m* ‘лиса-PL-POSS.1SG, *loga-ní-dɛt* ‘лиса-PL-POSS.3PL’;
Макарий (Нар. нар.): *ку-т* ‘человек-PL’, *калита-т* ‘подмышка-PL’;

Центр. XX в.
ЛЯНС (нар.), Беккер и др. [5] (Ср.-об. чум.), Быконя и др. [17] (нар.): (Лас.) *qú-la* ‘человек-PL’, (Лас.) *téw-la* ‘зуб-PL’, (Кен.) *hɨr-di-p* ‘корова-PL-ACC’, (Кен.) *kɨw-lʲika-t* ‘капля-DIM-PL’, (Кен.) *tebelqu-la-t* ‘мужчина-PL-PL’, (Кен.) *qu-la-n eː-ɣ-aːdət* ‘человек-PL-PL быть-PST-3PL’, (Кен.) *kɨbamar-la-n-d-e* ‘ребенок-PL-GEN-POSS.3SG-COM’, (Соис.) *tɨmnʲa-sə-t* ‘брат-RECIP-PL’, (Соис.) *nʲanʲa-ssə-də-nan* ‘сестра-RECIP-PL-LOC2.ABL’; (?нар.) *timńa-se-la* ‘брат-RECIP-PL’;

Беккер и др. [5] (Ср.-об. чум.): (Тюх.) *əlma ̈-ni ̈-l* ‘ребенок-PL-POSS.2SG’;
ЛЯНС (вас.): (Воль.) *loɣo-t* ‘черт-PL’, (Воль.) *kobɨ-n e-j-adɨt* ‘шкура-PL быть-EP-3PL’, (Калг.) *nenga-i-m* ‘сестра-PL-POSS.1SG’, (Воль.) *tebńa-sa-t* ‘брат-RECIP-PL’, (Воль.) *ära-l-mi-t* ‘старик-ADJZ-COLL-PL / родители’, (Калг.) *ťebńa-ssi̯-t* ‘брат-COLL-PL’;

Беккер и др. [5: 59] (Вас. чум.): (Воль.) *qum-ni̯-dɨt* ‘человек-PL-POSS.1PL’;
Сабо (Нап. тым.): *quu-la* ‘человек-PL’, *čumbne ̈-la* ‘волк-PL’, *ii-la-d* ‘сын-PL-POSS.3SG’;
ЛЯНС (тым.): (Нап.) *korgo-t* ‘медведь-PL’, (Нап.) *kede-t* ‘кишка-PL’;

Беккер и др. [5: 59] (Тым. чум.): (Вандж.) *peqᵊ-i-l* ‘лось-PL-POSS.2SG’, (Нап.) *qoroɣo-ni̯-m* ‘медведь-PL-POSS.1SG’.
Основным суффиксом множественного числа в северных диалектах по материалам XIX–XXI вв. является *-t(V)* ‘PL’, который в данных XX–XXI вв. также имеет дополнительный основной аллофон *-n* (что в том числе зависит и от фонетического окружения – появляется перед гласными и сонорными согласными, хотя и не обязательно). В посессивных формах суффикс множественности отражен в данных XIX–XXI вв. в виде *-i* ‘PL’. С терминами родства зафиксированы только примеры за XX–XXI в. с суффиксами *-mɨt* / *-min* ‘COLL.PL’ и *-sɨt* ‘RECIP.PL’. В источниках за XX–XXI в. для выражения собирательной множественности с

одушевленными и неодушевленными существительными используется маркер *-mɨ* ‘COLL’ (без дополнительных показателей множественности), который противопоставляется суффиксу расчлененного множества *-t(V)* ‘PL’.

Сев. XIX в.

Кастрен (таз.): *loḳa-t (-n’a)* ‘лиса-PL’, *kana-t* ‘собака-PL’, *loḳa-í-l* ‘лиса-PL-POSS.2SG’, *kana-i-t* ‘собака-PL-POSS.3SG’, *êêdše-lj-m* ‘слово-ADJZ-COLL’, *eedie-lj-mы* ‘деревня-ADJZ-COLL’;

Кастрен (ел.): *ate-t* ‘олень-PL’, *kana-t* ‘собака-PL’;

Кастрен (баи.): *logá-t (-n)* ‘лиса-PL’, *šíba-t* ‘утка-PL’, *muuni-i-m* ‘палец-PL-POSS.1SG’, *loga-í-m* ‘лиса-PL-POSS.1SG’;

Кастрен (кар.): *kana-t* ‘собака-PL’, *kî-t* ‘река-PL’, *loḳa-í-m* ‘лиса-PL-POSS.1SG’, *iite-lj-mэ (-me-t)* ‘деревня-ADJZ-COLL(-PL)’;

Сев. XX–XXI в.

К&Б [16] (ср.-таз.): *nō-m/н* ‘дерево-PL’, *кана-т/н* ‘собака-PL’, *nō-ū-мы* ‘дерево-PL-POSS.1SG’, *кана-ū-мы* ‘собака-PL-POSS.1SG’, *апа-ль-мы-т* ‘отец-ADJZ-COLL-PL / родители’, *ӷумы-ль-мы* ‘человек-ADJZ-COLL, *nō-ль-мы* ‘дерево-ADJZ-COLL;

К&Б [16] (в.-таз.): *шūпа-н* ‘утка-PL’, *кочый-ū-м* ‘сустав-PL-POSS.1SG’, *ө̄тä-й-мы* ‘олень-ADJZ-COLL’;

К&Б [16] (в.-толь.): *руши-н* ‘русский-PL’;

ЛЯНС (Кел. ел.): *iɣa-t / iɣa-n* ‘ребенок-PL’, *čʲóbe-j-mɨ* ‘лист-ADJZ-COLL’, *qana-j-mɨ* ‘собака-ADJZ-COLL’, *imáqóte-j-mɨ-n* ‘бабушка-ADJZ-COLL-PL’;

К&Б [16 (Кел. ел.): *ӷө̄нä-н* ‘кет-PL’, *тə̄тыпы-т* ‘шаман-PL’;

ЛЯНС (баи.): *loka-t* ‘лиса-PL’;

К&Б [16] (баи.): *руши-т* ‘русский-PL’, *тімня-и-ты* ‘брат-PL-POSS.3SG’, *леммы-ль-мы* ‘доска-ADJZ-COLL’;

Казакевич [6] (1998–1999, 2003) (тур.): *lōzy-t* ‘черт-PL’, *tịmńa-sy-t* ‘брат-RECIP-PL’; *iɣá-t* ‘ребенок-PL’, *móadə-lʲ-mɨ* ‘дом-ADJZ-COLL’.

Отдельно стоит выделить существенное количество примеров на *-lat* в южной, переходной диалектной зонах и нарымском диалекте, где данное сочетание следует трактовать не как контаминацию *-la* ‘PL’ и *-t* ‘PL’, а как сочетание *-la* ‘PL’ и *-t* ‘POSS.3SG’: (Сон. ср.-об.) *wes tabín ku-lá-t* (PL-POSS.3SG) *poйzʲí jéqwat* ‘вся семья у него обута’ (досл. ‘все люди его с обувью есть’); (УО кет.) *amdə-la-t* (PL-POSS.3SG) *teban essat wargə i taːdə* ‘рога ее у нее были большие и прямые’.

4. Заключение

1) В результате исследования было выяснено, что основным суффиксом множественного числа в южной и переходной языковых зонах является *-la* ‘PL’, в центральных васюганском и тымском, а также северных диалектах – *-t(V)* ‘PL’. В центральных и северных диалектах у показателя *-t(V)* ‘PL’ имеется второй стандартный фонетический вариант в виде *-n*, появление которого зависит, в том числе, от фонетического окружения (появляется перед гласными и сонорными согласными).
2) В центральном нарымском диалекте встречаются оба показателя *-t(V)* ‘PL’ и *-la* ‘PL’ в том числе в контаминированной форме *-lat* (*-la-t* ‘PL-PL’) ‘PL’, где для северной части более характерен *-t(V)* ‘PL’, а для южной *-la* ‘PL’.
3) В северных и центральных диалектах в посессивных формах используется суффикс *-i (-ni)* ‘PL’, что представляет собой сохранение архаики, в то время как в южных и переходных материалах, а также частично в центральных данных за XX в. по нарымскому диалекту и тымскому идиому текстов Л. Сабо он был вытеснен показателем *-la* ‘PL’ в тех же позициях.

4) Во всех диалектах с терминами родства появляется суффикс взаимной связи *-sa-* ‘RECIP’ вместе с *-t(V)* ‘PL’ – *-sat* ‘RECIP.PL’; реже в южных, центральных и северных диалектах суффикс собирательного множества *-mɨ-* ‘COLL’ вместе с *-t(V)* ‘PL’ – *-mɨt* ‘COLL.PL’. Примечательно то, что в южных и переходном диалектах вместе с данными показателями *-sa-* ‘RECIP’ и *-mɨ-* ‘COLL’ в большинстве случаев сохраняется суффикс *-t(V)* ‘PL’, а не *-la* ‘PL’, что стоило бы ожидать. Несмотря на это, в верхнекетских и нарымских материалах за XX в. зафиксированы отдельные случаи с суффиксом *-la* ‘PL’ в своем составе: (в.-кет.) *-mɨtla* (*-mɨ-t-la* ‘COLL-PL-PL’), (нар.) *-sala* (*-sa-la* ‘RECIP-PL’).
5) В северных диалектах для выражения собирательного множества без дополнительных числовых показателей с одушевленными и неодушевленными существительными используется показатель *-mɨ* ‘COLL’, который противопоставляется суффиксу расчлененного множества *-t(V)* ‘PL’.

Сокращения

1) Вандж. – Ванджилькынак на Тыме, Воль. – Вольджа на Чижапке, Ив. – Иванкино на Оби, Калг. – Калганак на Чижапке, Кар. – Карелино на Кети, Кел. – Келлог на Елогуе, Кен. – Кенга на Кенге, Лас. – Ласкино на Оби, Мар. – Марково на Кети, Нап. – Напас на Тыме; Нар. – Нарым на Оби, Соис. – Соиспаево на Парабели, Сон. – сондровские говоры (Старо- и Новосондрово) на Оби, Тог. – Тогур на Оби, Тюх. – Тюхтерево на Оби, УО – Усть-Озерное на Кети.
2) баи. – баихенский, в.-кет. – верхнекетский, в.-таз. – верхнетазовский, в.-толь. – верхнетолькинский, вас. – васюганский, ел. – елогуйский, ив. – иванкинский, кар. – карасинский, кет. – кетский, н.-кет. – нижнекетский, н.-чаин. – нижнечаинский, нар. – нарымский, н.-кет. – нижнекетский, ср.-кет. – среднекетский, ср.-об. – среднеобской, ср.-таз. – среднетазовский, таз. – тазовский, тым. – тымский, тур. – туруханский, чаи. – чаинский, чул. – чулымский.
3) В.-кет. сюс., тюй. – верхнекетские сюсюкумы, тюйкумы, В.-об. сюс. – верхнеобские сюсюкумы, Вас. чум. – васюганские чумылькупы, Ср.-кет. сюс. – среднекетские сюсюкумы, Ср.-об. чум. – среднеобские чумылькупы, Тым. чум. – тымские чумылькупы, шеш. – шешкупы / шешкумы.
4) Пер. – переходный иванкинский диалект, Сев. – северные диалекты, Центр. – центральные диалекты, Юж. – южные диалекты.
5) Григ. – Григоровский, досл. – дословно, К&Б – Казакевич и Будянская, ЛЯНС – лаборатория языков народов Сибири, мат. – материалы, н.п. – населенный пункт, недост. примеров – недостаточно примеров, посс. – посессивный, ПУ – прауральский, соб. форма – собирательная форма, сов. однородн. предм. – совокупность однородных предметов, сущ. – существительное, табл. – таблица.
6) 1 – первое лицо, 2 – второе лицо, 3 – третье лицо, ACC – аккузатив, ADJZ – аджективайзер, COLL – суффикс собирательной множественности, COM – комитатив, DIM – диминутив, DU – двойственное число, EP – эпентеза, GEN – генитив, LOC2.ABL – локатив2-аблатив, PL – множественное число, POSS – посессивность, PST – прошедшее время, RECIP – суффикс взаимной связи / реципрок, SG – единственное число.

Список литературы / References

[1]. Осипова А. А. Некоторые способы образования множественного числа имен существительных в селькупском языке (На материале говора пос. Келлог) // Происхождение аборигенов Сибири и их языков. Материалы межвузовской конференции 14–16 июня 1973 года. Томск, 1973. – С. 75–76. / Osipova A. A. Some ways of forming the plural of nouns in the Selkup language (Based on the subdialect of the village of Kellog) // The origin of the aborigines of Siberia and their languages. Materials of the interuniversity conference on June 14-16, 1973. Tomsk, 1973. – pp. 75–76 (in Russian).

[2]. Кузнецова А. И., Хелимский Е. А., Грушкина Е. В. Очерки по селькупскому языку. Тазовский диалект. Том 1. М., 1980 / Kuznetsova A. I., Helimsky E. A., Grushkina E. V. Sketches of the Selkup language. Taz dialect. Volume 1. Moscow, 1980. (in Russian).

[3]. Беккер Э. Г. Грамматические категории имени существительного в южных диалектах селькупского языка: дисс. ... доктора филол. наук: 10.02.07. – Томск, 1984. – 466 с. / Bekker E. G. Grammatical categories of the noun in the southern dialects of the Selkup language: Thesis ... doctor of philological sciences: 10.02.07. – Tomsk, 1984. – 466 p. (in Russian).

[4]. Максимова Н. П. Морфологический способ выражения множественности в южных диалектах селькупского языка // Вопросы енисейского и самодийского языкознания. Томск, 1983. – С. 103–110 / Maksimova N. P. Morphological ways of expressing plurality in the southern dialects of the Selkup language // Issues of Yenisei and Samoyed linguistics. Tomsk. 1983. – pp. 103–110 (in Russian).

[5]. Беккер Э. Г., Алиткина Л. А., Быконя В. В., Ильяшенко И. А. Морфология селькупского языка. Южные диалекты. Ч. 1. Томск, 1995. – 292 с. / Bekker E. G., Alitkina L. A., Bykonya V. V., Ilyashenko I. A. Morphology of the Selkup language. Southern dialects. Part 1. Tomsk, 1995. – 292 p. (in Russian).

[6]. Казакевич О. А. Две женщины, семь теснин и тридцать воинов (О выражении квантитативности в фольклорных текстах северных селькупов) // Логический анализ языка. Квантитативный аспект языка. М.: Индрик, 2005. – С. 384–399 / Kazakevich O. A. Two women, seven gorges and thirty warriors (On the expression of quantification in the folklore texts of the Northern Selkups) // Logical analysis of the language. The quantitative aspect of language. M.: Indrik, 2005. – pp. 384–399 (in Russian).

[7]. Lingvodoc. Доступно по ссылке: http://lingvodoc.ispras.ru (дата обращения 27.03.2025).

[8]. Alatalo Jarmo (2024). Manuscripta Castreniana Ostiak-Samoiedica (MCOS). Available at: https://www.sgr.fi/manuscripta/ostiaksamoiedica (accessed: 27.03.2025).

[9]. Григоровский Н. П. Азбука сюссогой гулани / Сост. Н. П. Григоровским для инородцев Нарымского края. Издание православного миссионерского общества. Казань, 1879 / Grigorovsky N. P. ABC book for Sjussogoj gulani / Compiled by N. P. Grigorovsky for the natives of the Narym Krai. Publication of the Orthodox Missionary Society. Kazan, 1879. (in Russian).

[10]. Григоровский Н. П. Объяснение праздников Св. Церкви. На остяцко-самоедском языке / Сост. Н. П. Григоровским для инородцев Нарымского края. Казань, 1879. / Grigorovsky N. P. Explanation of the holidays of the Holy Church. In the Ostyak-Samoyed language / Compiled by N. P. Grigorovsky for the natives of the Narym Krai. Kazan, 1879. (in Russian).

[11]. Григоровский Н. П. Священная история. На остяцко-самоедском языке / Сост. Н. П. Григоровским для инородцев Нарымского края. Казань, 1879 / Grigorovsky N. P. Sacred History. In the Ostyak-Samoyed language / Compiled by N. P. Grigorovsky for the natives of the Narym Krai. Kazan, 1879. (in Russian).

[12]. Григоровский Н. П. Молитвы и о сердечной молитве к Богу. На остяцко-самоедском языке / Сост. Н. П. Григоровским для инородцев Нарымского края. Казань, 1879 / Grigorovsky N. P. Prayers and about the heartfelt prayer to God. In the Ostyak-Samoyed language / Compiled by N. P. Grigorovsky for the natives of the Narym Krai. Kazan, 1879. (in Russian).

[13]. Макарий (Невский), еп. Материалы для ознакомления с наречием остяков Нарымского края, 1887 г. // Приложение № 7 к отчету об Алтайской и Киргизской миссиях за 1887 г. / Makary (Nevsky), bish. Materials for familiarization with the dialect of the Ostyaks of the Narym Krai, 1887 // Appendix No. 7 to the report on the Altai and Kyrgyz missions for 1887. (in Russian).

[14]. Макарий (Невский), еп. Беседы об истинном Боге и истинной вере на наречии обских остяков. Томск, 1900 / Makary (Nevsky), bish. Conversations about the true God and the true faith in the dialect of the Ob Ostyaks. Tomsk, 1900. (in Russian).

[15]. Szabó L. Selkup texts with phonetic introduction and vocabulary // Uralic and Altaic Series. Bloomington, 1967. 62 p.

[16]. Казакевич О. А., Будянская Е. М. Диалектологический словарь селькупского языка (северное наречие) Екатеринбург: Институт филологии СО РАН, 2010 / Kazakevich O. A., Budyanskaya E. M. Dialectological dictionary of the Selkup language (Northern dialect) Yekaterinburg: Institute of Philology SB RAS, 2010 (in Russian).

[17]. Быконя В. В., Кузнецова Н. Г., Максимова Н. П. Селькупско-русский диалектный словарь. Томск: ТГПУ, 2005. – 348 с. / Bykonya V. V., Kuznetsova N. G., Maksimova N. P. Selkup-Russian dialect dictionary. Tomsk: TSPU, 2005. – 348 p. (in Russian).

[18]. Глушков С. В., Байдак А. В., Максимова Н. П. Диалекты селькупского языка // Селькупы: очерки традиционной культуры и селькупского языка. – 2-е изд., испр. и доп. – Томск: Национальный исследовательский Томский политехнический университет, 2013. – С. 49–63 / Glushkov S. V.,

Baydak A.V., Maksimova N. P. Dialects of the Selkup language // Selkups: essays on traditional culture and the Selkup language. – 2nd ed., ispr. and add. – Tomsk: National Research Tomsk Polytechnic University, 2013. – pp. 49-63 (in Russian).
[19]. Collinder B. Comparative Grammar of the Uralic Languages (Handbook of the Uralic Languages: Part 3). Stockholm: Almqvist and Wiksell. 1960.
[20]. Castren M. A. Grammatik der Samojedischen Sprachen. St. Petersburg: Kaiserliche Akademie der Wissenschaften. 1854.
[21]. Donner K. Siperia: elämä ja entisyys. Helsinki 1933.
[22]. Joki A. 1936 Eine Untersuchung über das Object in der uralischen Sprachen: Finisch-ugrische Forschungen. Band XXXII, Heft 3, 1956. – S. 1–41.
[23]. Künnap A. System und Ursprung der kamassischen Flexionssuffixe. Helsinki. 1971.
[24]. Lehtisalo T. Uber die Primaren ururalischen Ableitungssuffixe Helsinki: Suomalais-ugrilainen seura, 1936.

Информация об авторах / Information about authors

Сергей Васильевич КОВЫЛИН – кандидат филологических наук, старший научный сотрудник лаборатории «Лингвистические платформы» Института системного программирования с 2019 года; доцент кафедры языков народов Сибири Томского государственного педагогического университета с 2019 года. Сфера научных интересов: корпусная лингвистика, компьютерная лингвистика, документация и ревитализация исчезающих языков и культур.

Sergei Vasilievich KOVYLIN – Cand. Sci. (Philol.), Senior research fellow of the laboratory “Linguistic platforms” of the Institute for System Programming of the RAS since 2019; Associate professor of the Department of Indigenous Siberian Languages at Tomsk State Pedagogical University since 2019. Research interests: corpus linguistics, computational linguistics, documentation and the revitalization of endangered languages and cultures.

DOI: 10.15514/ISPRAS-2025-37(6)-12

Тюрко-монгольские параллели в лексике материальной культуры тюркских языков Урало-Поволжья (на материале названий мастей лошади)

Р.Т. Муратова, ORCID: 0000-0003-4223-0675 <bairima@yandex.ru>

Институт истории, языка и литературы
Уфимского федерального исследовательского центра Российской академии наук,
Россия, 450054, г. Уфа, пр. Октября, д. 71.

Аннотация. В статье рассматриваются названия мастей лошади, которые имеют лексические параллели в тюркских языках Урало-Поволжья и монгольских языках. Исследования проводились с учетом данных по этимологии и лексикологии тюркских и монгольских языков. Предпринималась попытка выявления ареалов распространения отдельных лексем. Поиск этимологий и картографирование проводились на лингвистической платформе ЛингвоДок. Выявлены следующие особенности тюрко-монгольских параллелей цветообозначений в тюркских языках Урало-Поволжья: во-первых, часть общих для тюркских языков Урало-Поволжья и монгольских языков лексем для обозначения цвета являются родственными на генетическом уровне, происхождение которых восходит к праалтайским формам; во-вторых, в монгольских языках встречаются тюркские, в тюркских языках монгольские заимствованные названия цвета. Установлено, что в тюркских языках Урало-Поволжья монгольские заимствованные цветообозначения являются главным образом названиями мастей лошади, в то время как в монгольских языках тюркские заимствования могут обозначать как масть животных, так и цвет в целом.

Ключевые слова: тюркские языки Урало-Поволжья; тюрко-монгольские лексические параллели; лексика материальной культуры; названия мастей лошади; лингвистическое картографирование; лингвистическая платформа ЛингвоДок.

Для цитирования: Муратова Р.Т. Тюрко-монгольские параллели в лексике материальной культуры тюркских языков Урало-Поволжья (на материале названий мастей лошади). Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 193–202. DOI: 10.15514/ISPRAS–2025–37(6)–12.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-28-01562. Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П. Иванникова РАН – ЦКП ИСП РАН.

Turkic-Mongolian parallels in the vocabulary of the material culture of the Turkic languages of the Ural-Volga region (based on the names of horse colors)

R.T. Muratova, ORCID: 0000-0003-4223-0675 <bairima@yandex.ru>

Institute for History, Language and Literature
Ufa Federal Research Centre of Russian Academy of Sciences,
71, prospekt Oktyabrya, Ufa, 450054, Russia.

Abstract. The article discusses the names of horse breeds that have lexical parallels in the Turkic languages of the Ural-Volga region and the Mongolian languages. The research was conducted based on data on the etymology and lexicology of the Turkic and Mongolian languages. An attempt was made to identify the distribution areas of individual lexemes. The search for etymologies and mapping were conducted using the linguistic platform LingvoDoc. The following features of the Turkic-Mongolian parallels of color designations in the Turkic languages of the Ural-Volga region have been identified: first, some of the color designations common to the Turkic languages of the Ural-Volga region and the Mongolian languages are genetically related and originate from the Proto-Altaic forms; second, the Mongolian languages contain Turkic and the Turkic languages contain Mongolian borrowed color names. It has been established that in the Turkic languages of the Ural-Volga region, Mongolian borrowings (color names) are mainly names of horse colors, while in the Mongolian languages, Turkic borrowings can refer to both animal's colors and colors.

Keywords: Turkic languages of the Ural-Volga region; Turkic-Mongolian lexical parallels; vocabulary of material culture; names of horse colors; linguistic mapping; linguistic platform LingvoDoc.

For citation: Muratova R.T. Turkic-Mongolian parallels in the vocabulary of the material culture of the Turkic languages of the Ural-Volga region (based on the names of horse colors). Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6,part 1, 2025, pp. 193-202 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-12.

Acknowledgements. The reported study was granted by Russian Science Foundation, project no. 24-28-01562. The results were obtained using the services of the Ivannikov Institute for System Programming (ISP RAS) Data Center.

1. Введение

Ученые насчитывают в лексике тюркских и монгольских языков до 25 % общих элементов [1: 351]. Одной из проблем монголистики и тюркологии является разграничение общего наследия, древних и более поздних заимствований. Другими словами, при изучении лексических параллелей, встречающихся в двух языковых семьях, необходимо иметь в виду, что они могут либо восходить к общим алтайским корням, либо являться древними (на пратюркском, прамонгольском уровне) или более поздними заимствованиями в тюркских / монгольских языках. В данной статье рассмотрим названия мастей лошади, которые имеют лексические параллели в тюркских языках Урало-Поволжья и монгольских языках, и попытаемся выяснить среди них лексемы общего наследия и заимствованные названия.

2. К вопросу о лексических элементах общего наследия и заимствованиях в тюркских и монгольских языках

2.1 О лексических элементах общего наследия

Лексические элементы общего наследия в тюркских и монгольских языках связываются с алтайской гипотезой. Гипотеза о характере родства алтайских языков, в частности, тюркских и монгольских, была выдвинута в XVIII в. Ф.И. Страленбергом, который «представил первую классификацию уральских (угро-финских и самодийских) и алтайских (тюркских, монгольских и тунгусских) языков» [2: 29].

Позже урало-алтайская теория в различных ее вариантах получила признание у многих исследователей уральских и алтайских языков, которые рассматривают фонетические и лексические соответствия в разных языках как наследие от одного языка-основы, то есть от общего предка [3-7].

В.М. Иллич-Свитыч отмечал весьма отдаленное родство трех алтайских групп – тюркской, монгольской и тунгусо-маньчжурской [8]. В.И. Цинциус впервые высказала мысль о возможности реконструкции лексики алтайских языков по лексико-семантическим группам [9].

Одной из самых трудных проблем в этом вопросе является разграничение общего наследия и древних заимствований. В связи с этим А. Рона-Таш приводит несколько аргументов, среди которых наиболее важными являются следующие:

- этимологический аргумент (если слово встречается в языках А и Б в регулярно соответствующих формах, но в языке А для него нет этимологии, тогда как его основа может быть найдена в языке Б, то это, скорее всего, заимствование в языке А из языка Б);
- семантико-исторический аргумент (если слово встречается в языках А и Б в регулярно соответствующих формах, но имеет только одно конкретное или специализированное значение в языке А и в то же время гораздо более широкий круг значений в языке Б, то, вероятнее всего, что язык А заимствовал его из языка Б, хотя, возможно, что вторичное ограничение значения могло произойти на почве языка А);
- аргумент основного словарного фонда (чем больше соответствий может быть найдено в основном словарном фонде языков А и Б, тем больше возможность их генетического родства);
- культурно-исторический аргумент (если данный комплекс терминов культуры, экономики и социальной истории совпадает в языках А и Б и если этот комплекс начал существовать позже, то можно предположить, что рассматриваемая терминология была заимствована или в языке А, или в языке Б, или же в обоих этих языках) [10].

А.В. Дыбо, подчеркивая сложность изучения пратюркско-монгольских заимствований, отмечает, что алтайская этимология еще не настолько разработана, чтобы отделять такие заимствования от слов, представляющих собой алтайские этимологические параллели: «К пратюркским заимствованиям в прамонгольский язык – из лексики, которая не допускает интерпретации как заимствованная из древнетюркского в старомонгольский или среднемонгольский – могут относиться лишь те случаи, когда рассматриваемое прамонгольское слово содержит характерный тюркский морфологический / словообразовательный признак (доказать наличие таковых при, мягко говоря, неполном описании монгольской исторической морфонологии и словообразования нелегко) или для пратюркского слова имеется альтернативное монгольское сближение, для которого гипотеза о заимствовании менее вероятна из-за сильных поверхностных фонетических различий. Прамонгольские заимствования в пратюркский должны удовлетворять тем же критериям» [11: 181–182].

Отметим, что А.В. Дыбо впервые разработала методику семантической реконструкции наименований частей тела (плечевой пояс), одним из ценных достижений которой являются номинационные решетки по материалам тюркских, монгольских, тунгусо-маньчжурских языков с опорой на ностратическую теорию. Суть её исследования состоит в том, что «реконструкция лексики праязыка должна включать в реконструкцию первоначальных значений слов и объяснение изменений этих значений» [12].

В.И. Рассадин также предлагает рассматривать монгольские заимствования в различных тюркских языках как результат древнейших и разнообразных контактов алтайских народов [13].

2.2 О заимствованной лексике в тюркских и монгольских языках

Заимствованный пласт лексики также требует пристального внимания. Различаются ранние и поздние заимствования. Считается, что наиболее ранние из предполагаемых монгольских заимствований в тюркские языки оказываются заимствованными в общетюркский язык после отпадения булгарской ветви. Ранние монгольские заимствования в основном характерны для всех подгрупп тюркских языков.

Кроме общих для всех тюркских языков монголизмов, в тех или иных языках имеются более поздние заимствования. Например, анализ общих лексических единиц в башкирском и монгольском языках позволил Э.Ф. Ишбердину выделить несколько критериев, способствующих выявлению заимствований. Одним из важных критериев выделения поздних заимствований является определение степени распространения общих лексических единиц в башкирских говорах, в современных тюркских языках, а также в древних письменных памятниках. Для определения степени ареальной распространенности монголизмов Э.Ф. Ишбердин выбрал лингвогеографический метод. Нанесение на карты некоторых слов, общих для башкирского и монгольского языков, позволило сделать вывод о том, что «эти поздние заимствования в башкирских говорах распространены в основном в южных и восточных зонах территории, заселенной башкирами» [14: 32].

Исследователем отмечаются взаимодействие и взаимовлияние тюркских и монгольских народов: «Предварительное сравнение общих для тюркских и монгольских языков лексических единиц показывает, что из тюркских языков башкирский, татарский, казахский, киргизский, каракалпакский и в своеобразной форме чувашский стоят обособленно по степени усвоения монгольских слов. Кроме общих для всех тюркских языков монголизмов, в указанных языках имеются явные заимствования, видимо, более поздние, из монгольского и калмыцкого языков. Это могло быть результатом влияния монгольского языка в период монголо-татарского нашествия в XIII–XIV вв.» [14: 33].

Таким образом, несмотря на сложность и труднорешаемость вопроса разграничения общеалтайской лексики и древних заимствований, ученые делают небезуспешные попытки выявления общего и заимствованного пласта лексики в тюркских и монгольских языках.

3. Тюрко-монгольские параллели цветообозначений в тюркских и монгольских языках

3.1 Названия, родственные на генетическом уровне

Необходимо отметить, что среди цветообозначений также много общих тюркских и монгольских слов: одни из них – названия, родственные на генетическом уровне, другие – заимствования из монгольских в тюркских языках или заимствования из тюркских в монгольских языках.

К родственным словам на генетическом уровне можно отнести следующие цветообозначения, встречающиеся в тюркских и монгольских языках в близких значениях и восходящие к алтайским праформам:

- ПА **kàru* ‘черный’: ПТю. **Kara* ‘черный’ / ПМо. **kara* ‘черный’;
- ПА **s̯ājri* ‘белый, желтый’: ПТю. **siārïg* ‘желтый, белый’ / ПМо. **sïra* ‘желтый’;
- ПА **kə́k‘e* ‘синий, зеленый’: ПТю. **gə̄k* ‘синий, зеленый (макросиний)’ / ПМо. **köke* ‘синий, зеленый’;
- ПА **kòŋa* ‘коричневый, черный’: ПТю. **Koŋur* ‘бурый’ / ПМо. **koŋ-* ‘светло-коричневый;
- ПА **ā́lV* ‘пестрый’: ПТю. **āla* ‘пестрый’ / ПМо. **ala-g* ‘пестрый’;
- ПА **boŕV* ‘серый’: ПТю. **boŕ* ‘серый’ / ПМо. **boro* ‘серый’;

- ПА **šop'é* ‘веснушки, пятна’: ПТю. **čopur, *čap-* ‘пестрый, рябой’, ‘непородистый, неряшливый’, ‘высыпать (о сыпи, чирьях)’ / ПМо. **čob, *čow-kur* ‘пятнышко, веснушка; пятнистый, пестрый’ [15-16].

3.2 Тюркские заимствования в монгольских языках

К тюркским заимствованиям в монгольских языках относятся следующие цветообозначения:

- ПТю. **jęgre-n* ‘рыжий (о масти лошади); олень, джейран, антилопа’ > монг.: халх.-монг. *зээрд,* бур. *зээрдэ,* калм. *зеерд,* даг. *джэ̄рдэ* ‘рыжий (о лошади)’ / халх.-монг. *зээрэн,* бур. *зээрэн,* калм. *зеерн,* даг. *джэ̄рэн* ‘антилопа, серна, джейран’;
- ПТю. **Kula* ‘желтый, саврасый’ > монг.: халх.-монг., калм. *хул* ‘саврасый’, бур. *хула* ‘саврасый’;
- ПТю. **čĀl* ‘серый, седой’ > монг.: калм. *цал,* халх.-монг. *цал буурал* ‘седой, чалый’;
- ПТю. **Kuba / *Koba* ‘желтоватый, сероватый’ > монг.: халх.-монг., бур. *ухаа, хуа, хуаа* ‘каурый (о масти лошади); светло-коричневый (о цвете)’; калм. *хо, хоо* ‘светло-жёлтый, соловый (о масти лошади); коричнево-желтый (о цвете)’;
- ПТю. **Āl* ‘алый, розовый’ > монг.: калм. *al* ‘уст. светло-красный’, халх.-монг. *ал* ‘красный’;
- ПТю. **čakïr* ‘светло-серый, серовато-голубой’ > монг.: халх.-монг. *цэхэр,* бур. *сэхир* ‘светло-серый’; бур. *сахир,* калм. *цәкр* ‘белесый, бледный’ [15-16].

3.3 Монгольские заимствования в тюркских языках Урало-Поволжья

Относительно заимствований из монгольских языков в тюркские А.М. Щербак отмечает, что «речь должна идти главным образом об относительно поздних заимствованиях (после XIII в.) из монгольских языков в тюркские» [17: 29].

В тюркских языках монгольские заимствования распространены неравномерно: самое большое количество монголизмов выявлено в сибирских тюркских языках (якутский, тувинский, алтайский, хакасский, шорский); далее – в кыпчакских языках арало-каспийского ареала (киргизский, казахский, каракалпакский, ногайский); в кыпчакских языках Урало-Поволжья (башкирский, татарский) их меньше, чем в первых двух группах [13: 77; 18: 334-335].

Цветообозначения в тюркских языках Урало-Поволжья, заимствованные из монгольских языков, отражают тесную взаимосвязь между тюркскими и монгольскими народами и составляют в основном названия мастей лошади.

В тюркских языках Урало-Поволжья следующие цветообозначения / названия мастей лошади являются монголизмами:

- башк., тат. *көрән,* чув. *кĕре, кĕрен* (тат. заимств.) ‘коричневый, бурый’, башк. диал. ‘фиолетовый’ < монг. < ПМо. **küre- (*küri)* ‘коричневый, бурый’ < ПА **k'iūŕu* ‘красный; коричневый, темный’ (*отметим, что эта лексема встречается во всех подгруппах тюркских языков*);
- башк., тат. *сапмар,* тат. *чапмар,* чув. *чупмар* ‘игреневый’ (тат. заимств.) < монг. < ПМо. **čabidar* ‘желтоватый, игреневый’ < ПА **č'ira* ‘серый’ (*эта лексема встречается также в кыпчакских, киргизско-кыпчакских и уйгуро-огузских подгруппах тюркских языков*);
- башк. *бурыл* ‘чалый, серый’, тат. *бурлы* ‘чалый, серый’, чув. *пурлă* ‘бурый, чалый’, *пăвăрлă* ‘сероватый, чалый, буланый’ < монг. < ПМо. **buɣurul* ‘серый, седой’ < ПА **bagu* ‘белый, серый’(*эта лексема встречается во всех подгруппах тюркских языков*);

- башк., тат. *бүртә* ‘караковый’ < позднемонг. *börtü* ≈ *börte* ‘пестрый, с пятнами, пестрый, в крапинку; серый’ (*эта лексема встречается также в сиб.-тат., кирг, уйг. языках*);
- башк., тат. *кир* ‘мухортый’ < монг. < ПМо. **kehere* (*эта лексема встречается во всех подгруппах тюркских языков, кроме булгарской*);
- башк. *тарлан* ‘серый, сивый с примесями другого цвета’, тат. *тарлан* чёрно-пятнистый, сивый с жёлтыми пятнами (о лошади) < монг. *тарлан* ‘пестрый, крапчатый; пятнистый’ (о крупном рогатом скоте, птицах) (*эта лексема также встречается во всех подгруппах тюркских языков, кроме булгарской и огузской*).

Итак, в тюркских языках Урало-Поволжья из 6 монгольских заимствований 5 наименований являются названиями мастей лошади. Вместе с тем название неспектрального цвета *көрән* ‘коричневый’, которое входит в активный словарный запас этих языков, изначально также обозначало бурую масть лошади.

Из всех монголизмов-цветообозначений в тюркских языках Урало-Поволжья три лексемы встречаются во всех тюркских языках Урало-Поволжья, два из которых в чувашском являются заимствованием через татарский язык. Три названия масти лошади встречаются только в кыпчакских языках Урало-Поволжья. Этот факт, во-первых, подтверждает предположения ученых о том, что монголизмы в тюркских языках в основном появились после отделения булгар (так как в чувашском языке они являются заимствованиями через татарский язык); во-вторых, позволяет рассуждать о более раннем характере заимствований, связанный со скотоводческим образом тюрко-монгольских контактов, тогда как более поздние монгольские заимствования в тюркских языках других ареалов являются названиями цвета вообще (напр., алт. *ногон,* тув. *ногаан,* хак. *ноған* ‘зеленый’ < монг. < ПМо. **nogoγan* ‘зеленый’).

4. Монголизмы в тюркских языках Урало-Поволжья на фоне других тюркских языков (на примере названия масти көрән ‘бурый’)

Монголизмы в тюркских языках Урало-Поволжья характерны для всех тюркских языков или представляют собой ареальное явление? Попытаемся ответить на этот вопрос на примере лексемы *көрән* ‘коричневый, бурый’, разместив на карте ареалы ее распространения в тюркских языках.

В современных тюркских языках и их диалектах слово *көрән* наблюдается в следующих фонетических вариациях: башк., тат., сиб.-тат. *көрән*, ног. *куьрен*, аз. *kürän*, каз. *күрең*, алт., туб., куманд. *күреҥ*, телеут. *күрүң,* хак., шор. *күрең*, уйг. *күрән*, кирг. *күрөң*, кум. *гюрен,* ккалп. *гүреңь*, тур. диал. *küren,* туркм. диал. *күрөң,* узб. диал. *күрән*, тув. *хүреңь,* тоф. *hүреңь,* як. *күрэҥ* / *күрүҥ*, чув. *кӗре, кӗрен* (тат. заимств.).

Во всех языках слово имеет значение – ‘масть лошади (бурый, карий, гнедой, рыжий, саврасый)’. В башк., тат., сиб.-тат., хак., тув., шор., алт. языках данное слово также употребляется как основная лексема для обозначения коричневого цвета.

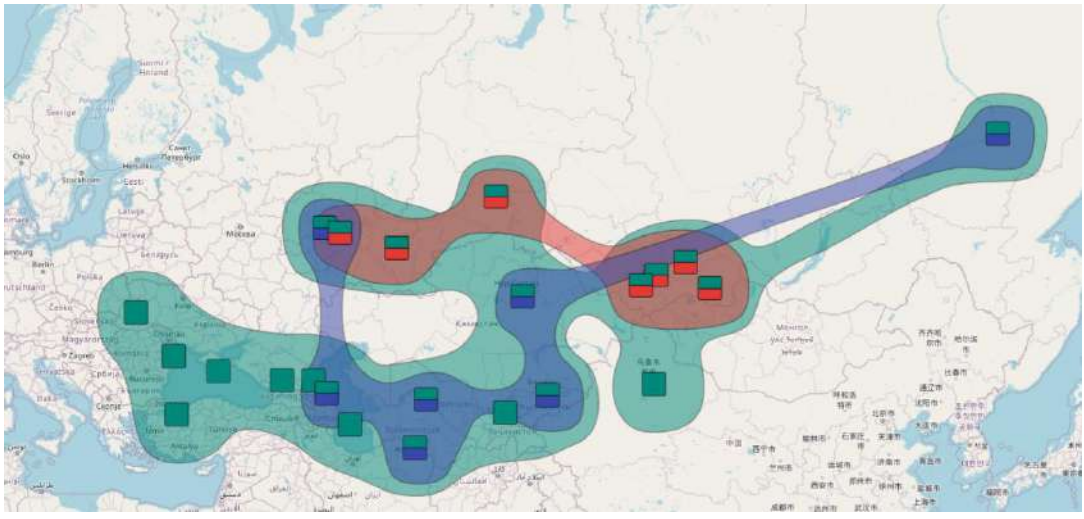
В ряде тюркских языков (каз., ккалп., кум., кирг., туркм., як.) основной лексемой для коричневого цвета являются рефлексы ПТю. **Koŋur,* в них *küren* употребляется лишь для обозначения масти лошади.

Отметим на карте те языки, в которых лексемами-доминантами для обозначения коричневого цвета являются рефлексы ПТю. **Koŋur* или монгольское заимствование *küren* (карта 1).

Карта показывает, что рефлекс пратюркской лексемы **Koŋur* ‘бурый’ в качестве лексемы-доминанты для коричневого употребляется в чувашском, якутском, а также кыпчакских языках Средней Азии, монгольское *küren* – в кыпчакских языках Урало-Поволжья и сибирских тюркских языках (кроме якутского).

В остальных тюркских языках – в карлукских и огузских и некоторых кыпчакских языках Кавказа, которые находятся по краям ареала распространения тюркских языков, обе лексемы не являются основными для обозначения коричневого цвета. В этом ареале коричневый цвет представлен другими лексемами, например: ног. *моры* (< возможно, инд.-евр.), аз. *darçını* ‘*букв.* цвета корицы’, тур. *kahverengi* ‘*букв.* цвета кофе’, узб. *жигар ранг* ‘темно-коричневый, *букв.* цвета печени’, уйг. *беғирән* ‘*букв.* цвета печени’.

Таким образом, лексема *көрән* в разных фонетических вариантах распространена во всех подгруппах тюркских языков (в чувашском – заимствование из тат.). Вместе с тем, *көрән* как лексема-доминанта для обозначения коричневого цвета получило широкое распространение в кыпчаских языках Урало-Поволжья и сибирских тюркских языках, оставляя за исконном тюркским словом **Koŋur* ‘бурый’ обозначение мастей животных (кроме лошади) и цвета некоторых других предметов (например, цвета глаз, волос, кожи).



Условные обозначения:

- ареал распространения тюркских языков
- ареал распространения лексемы-доминанты **Koŋur* ‘бурый’ для обозначения коричневого цвета в тюркских языках (ккалп. *қоңыр*, каз. *қоңыр*, кирг. *коңур*, кум. *къонгур*, туркм. *гоңур*, як. *хоҥор*, чув. *хăмăр*)
- ареал распространения лексемы-доминанты *küren* ‘коричневый’ для обозначения коричневого цвета в тюркских языках (башк. *көрән*, тат. *көрән*, сиб.-тат. *көрән*, алт. *күреҥ*, хак. *күрең*, тув. *хүрең*, шор. *күрең*)

*Карта 1. Распространение лексем-доминант küren ‘коричневый’ и *Koŋur ‘бурый’ для обозначения коричневого цвета в тюркских языках (карта составлена на платформе Лингводок [lingvodoc.ispras.ru]).*

*Cart. 1. The distribution of dominant lexemes kuren ‘brown’ and *Koŋur ‘brown’ to denote brown in the Turkic languages (the map was created on the Lingvodok platform [lingvodoc.ispras.ru]).*

5. Заключение

Итак, мы можем наблюдать следующие особенности тюрко-монгольских паралеллей цветообозначений в тюркских языках Урало-Поволжья:

- Во-первых, часть общих для тюркских языков Урало-Поволжья и монгольских языков лексем для обозначения цвета, являются родственными на генетическом уровне, происхождение которых восходит к праалтайским формам. Это связано с

тем, что цветообозначения восходят к более глубокому, протоязыковому состоянию общетюркского праязыка, когда происходило формирование многих семантических и формальных категорий.

- Во-вторых, в монгольских языках встречаются тюркские, в тюркских языках монгольские заимствованные названия цвета. В тюркских Урало-Поволжья монгольские заимствования-цветообозначения являются главным образом названиями мастей лошади, в то время как в монгольских языках тюркские заимствования могут обозначать как масть лошади, так и цвет в целом.

То есть, мы на примере тюрко-монгольских параллелей цветообозначений в тюркских языках Урало-Поволжья, основываясь на этимологических сведениях и лингвогеографических данных, полученнных в результате обработки на платформе ЛингвоДок, подтвердили предположения ученых о том, что монголизмы в тюркских языках в основном появились после отделения булгар (так как в чувашском языке они являются заимствованиями через татарский язык). Также выявлено, что монголизмы-цветообозначения в тюркских языках Урало-Поволжья являются главным образом названиями мастей лошади, что дает возможность предположить древний характер заимствований, связанный со скотоводческим кочевым образом тюрко-монгольских контактов.

Дополнительные сведения о проведенном исследовании можно получить на сайте Российского научного фонда [19].

Список сокращений

диал.	диалект	инд.-евр.	индоевропейские
аз.	азербайджанский	ПТю.	пратюркский
алт.	Алтайский	сиб.-тат.	сибирскотатарский
башк.	башкирский	тат.	татарский
бур.	Бурятский	телеут.	телеутский
гаг.	Гагаузский	тоф.	тофаларский
даг.	Дагурский	туб.	тубинский
калм.	Калмыцкий	тув.	тувинский
каз.	Казахский	тур.	турецкий
кирг.	Киргизский	туркм.	туркменский
ккалп.	каракалпакский	узб.	узбекский
кум.	Кумыкский	уйг.	уйгурский
куманд.	кумандинский	хак.	хакасский
монг.	монгольские	халх.-монг.	халха-монгольский
ног.	Ногайский	чув.	чувашский
ПА	праалтайский	шор.	шорский
ПМо.	прамонгольский	як.	якутский

Список литературы / References

[1]. Котвич В. Исследования по алтайским языкам. М., 1962. 371 с.
[2]. Баскаков Н.А. Алтайская семья языков и ее изучение. М., 1981. 135 с.
[3]. Владимирцов Б.Я. Работы по монгольскому языкознанию. М., 2005. 952 с.
[4]. Поливанов Е.Д. Статьи по общему языкознанию. М., 1968. 376 с.
[5]. Поливанов Е.Д. Труды по восточному и общему языкознанию. М., 1991. 623 с.
[6]. Поппе Н.Н. Урало-алтайская теория в свете советского языкознания // Известия АН СССР. Отделение литературы и языка. 1940. № 3.
[7]. Рамстедт Г.И. Введение в алтайское языкознание. М., 1957. 254 с.
[8]. Иллич-Свитыч В.М. Опыт сравнения ностратических языков (семитохамитский, картвельский, индоевропейский, уральский, дравидийский, алтайский). Введение. Сравнительный словарь (b–K). М., 1971. 370 с.
[9]. Цинциус В.И. Проблемы сравнительно-исторического изучения лексики алтайских языков //

Исследования в области этимологии алтайских языков. Л., 1979. с. 3–17.
[10]. Рона-Таш Л. Общее наследие или заимствования // Вопросы языкознания. 1974. № 2, с. 31–45.
[11]. Дыбо А.В. Лингвистические контакты ранних тюрков. Лексический фонд: пратюркский период. М., 2007. 223 с.
[12]. Дыбо А.В. Семантическая реконструкция в алтайской этимологии: Соматические термины (плечевой пояс). М., 1996. 390 с.
[13]. Рассадин В.И. Монголо-бурятские заимствования в сибирских тюркских языках. М., 1980. 115 с.
[14]. Ишбердин Э.Ф. Историческое развитие лексики башкирского языка. М., 1986. 152 с.
[15]. Starostin S.A., Dybo A.V., Mudrak O.A. An Etymological Dictionary of Altaic Languages. Leiden: Brill, 2003. 1556 p.
[16]. Дыбо А.В. Этимологический словарь базисной лексики тюркских языков. Астана, 2013. 616 с.
[17]. Щербак А.М. Тюркско-монгольские языковые контакты в истории монгольских языков. СПб., 2005. 195 с.
[18]. Малышева Н.В. Происхождение лексики живой природы: ономасиология и этимология (якутский язык и его диалекты). Диссертация ... доктора филол. наук. М., 2024. 503 с.
[19]. https://rscf.ru/project/WZWMHOTCvZoL_M4iqcaoPGh1pxUVkjZXRK1awWxYF_vAIRJYfXAdMvN5feFJpIx3MFpsf5eg/, дата обращения: 14.09.2025 г.

Информация об авторах / Information about authors

Римма Талгатовна МУРАТОВА – кандидат филологических наук, старший научный сотрудник отдела языкознания Ордена Знак Почета Института истории, языка и литературы Уфимского федерального исследовательского центра Российской академии наук. Сфера научных интересов: башкирский язык, тюркские языки, языки Урало-Поволжья, лексикология, этнолингвистика, компьютерная лингвистика, лингвогеография.

Rimma Talgatovna MURATOVA – Cand. Sci. (Philology), Senior Researcher, Department of Linguistics, Order of the Badge of Honor, Institute of History, Language and Literature, Ufa Federal Research Center, Russian Academy of Sciences. Area of scientific interests: Bashkir language, Turkic languages, languages of the Ural-Volga region, lexicology, ethnolinguistics, computer linguistics, linguistic geography.

DOI: 10.15514/ISPRAS-2025-37(6)-13

Material Culture Lexicon in M.A.Castrén's Dictionary (1844) and an Audio Dictionary of the Ižma Dialect (2012): Comparative Analysis on LingvoDoc

O.N. Bazhenova, ORCID: 0000-0002-5666-2260 <bazhenova-olga2011@mail.ru>
"Linguistic platforms" laboratory,
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. This article presents a comparative analysis of the material culture lexicon in the Izhma dialect of the Komi language, utilizing data from M.A. Castrén's 1844 dictionary (revised 2022 edition) and a 2012 audio dictionary of the Beloyarsk village dialect processed and uploaded by E.V. Kashkin onto the LingvoDoc platform. The study aims to identify key trends in lexical dynamics by examining the interplay of indigenous vocabulary with borrowings, innovations, and archaisms over a period of more than 160 years. The Izhma dialect, shaped by intensive historical contacts with Russian and Nenets languages in a unique multi-ethnic environment, offers a significant case for understanding language vitality and shift. The research employed the LingvoDoc platform for processing and analyzing 127 lexemes from Castrén's work and 167 from the modern audio dictionary. Lexemes were categorized into five thematic groups: dwelling, utensils and household items, clothing and footwear, tools and crafts, and transport. Each item was compared based on its presence in sources, phonological form, meaning, and etymology, identifying direct correspondences, archaization, innovation, lexical replacement, and phonetic-morphological changes. Results indicate distinct patterns across thematic groups. The "Tools and Crafts" category exhibited the highest proportion of archaisms (50%), reflecting the decline of traditional practices. Conversely, the "Transport" group showed the most significant innovation (73.9%), driven by new terminology and borrowings. The findings underscore that observed differences are not solely lexical transformations but also reflect varying recording completeness and focus between historical and modern sources. Overall, the Izhma dialect's material culture vocabulary reveals areas of both stability and active restructuring, providing valuable insights for reconstructing lexical subsystems, analyzing contact linguistics, and describing language evolution in peripheral Komi linguistic regions.

Keywords: Izhma dialect; Komi language; lexical comparison; material culture; lexical dynamics; language contact; LingvoDoc.

For citation: Bazhenova O.N. Material Culture Lexicon in M.A.Castrén's Dictionary (1844) and an Audio Dictionary of the Ižma Dialect (2012): Comparative Analysis on LingvoDoc. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 203-218. DOI: 10.15514/ISPRAS-2025-37(6)-13.

Acknowledgements. The research was funded by Russian Science Foundation № 25-78-20002 "Capabilities of Artificial Intelligence for Comparative-Historical Study of Low-Resource Languages of the Peoples of the Russian Federation". The results were obtained using the services of the Ivannikov Institute for System Programming (ISP RAS) Data Center.

Лексика материальной культуры в словаре М. А. Кастрена (1844) и аудиословаре ижемского диалекта говора (2012): сравнительный анализ по данным платформы LingvoDoc

О.Н. Баженова, ORCID: 0000-0002-5666-2260 <bazhenova-olga2011@mail.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Лаборатория 16.2 "Лингвистические платформы",
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Статья посвящена сравнительному анализу лексики материальной культуры ижемского диалекта коми языка. Исследование проведено на основе данных словаря М.А. Кастрена 1844 года (в переработанном издании 2022 года) и аудиословаря 2012 года говора с. Белоярск Приуральского района ЯНАО, обработанного и загруженного на платформу LingvoDoc Е.В. Кашкиным. Цель работы — выявить ключевые тенденции лексической динамики ижемского диалекта, изучая взаимодействие исконной лексики с заимствованиями, инновациями и архаизмами на протяжении более чем 160 лет. Особое внимание уделено ижемскому диалекту как уникальному случаю, сформировавшемуся под влиянием интенсивных исторических контактов с русским и ненецким языками в полиэтнической среде, что важно для понимания жизнеспособности языка и языковых сдвигов. Были проанализированы 127 лексем из словаря М. А. Кастрена и 167 лексем из современного аудиословаря говора с. Белоярск. Лексические единицы были категорированы по пяти тематическим группам: жилище, утварь и предметы быта, одежда и обувь, орудия труда и промыслов, а также транспорт. Каждая лексема сравнивалась по наличию в источниках, фонологической форме, значению и этимологии. В результате анализа были выявлены прямые соответствия, случаи архаизации, инновации, лексической замены и фонетико-морфологических изменений. Полученные результаты демонстрируют различные закономерности в исследуемых тематических группах. Наибольшая доля архаизмов (50%) зафиксирована в категории "Орудия труда и промыслов", что коррелирует с исчезновением обозначаемых реалий в связи с утратой традиционных промыслов. В то же время максимальное количество инноваций (73,9%) обнаружено в группе "Транспорт", что объясняется активным развитием новой арктической терминологии и заимствованиями. Отмечается, что не все различия между источниками являются исключительно результатом лексических изменений; некоторые могут быть связаны с неполнотой фиксации. Таким образом, лексика материальной культуры ижемского диалекта демонстрирует как зоны устойчивости, так и активной лексической перестройки. Полученные данные могут быть использованы для реконструкции лексических подсистем, изучения контактных влияний и описания языковой эволюции на периферии коми языкового ареала.

Ключевые слова: ижемский диалект; коми язык; лексическое сопоставление; материальная культура; лексическая динамика; языковые контакты; LingvoDoc.

Для цитирования: Баженова О.Н. Лексика материальной культуры в словаре М. А. Кастрена (1844) и аудиословаре ижемского диалекта говора (2012): сравнительный анализ по данным платформы LingvoDoc. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 203–218 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(6)–13.

Благодарности. Работа выполнена при поддержке гранта Российского Научного Фонда № 25-78-20002 «Возможности искусственного интеллекта для сравнительно-исторического изучения малоресурсных языков народов РФ». Результаты получены с использованием услуг Центра коллективного пользования Института системного программирования им. В.П. Иванникова РАН – ЦКП ИСП РАН.

1. Introduction

The Izhma dialect is one of the ten dialects of the Komi-Zyrian language, which belongs to the Permic branch of the Finno-Ugric language family [1-2]. Within the dialect continuum of the Komi language, the Izhma dialect occupies a special position. Its distinctive feature is a historically more intensive and prolonged contact with the Russian language compared to other Komi dialects. This long-term contact led to the formation of a mixed lexicon, which has been noted as one of the most prominent features of the Izhma dialect [2]. In the Izhma District of the Komi Republic, where a significant

portion of its speakers reside, the local idiom remains the primary code for daily communication. According to available data, over 75% of the district's population actively uses the Komi language in everyday life [2]. Such a high proportion of speakers indicates the preservation of the language in rural areas, although Russian dominates in urban settings [3].

The vocabulary of material culture holds particular historical and cultural significance for the Izhma dialect. The formation of the Izhma dialect by the end of the 18th century occurred in a unique multi-ethnic environment, including Komi, Russian settlers, and Nenets reindeer herders. This interaction led to the enrichment of the Izhma dialect's lexicon with borrowings from Russian and Nenets languages, distinguishing it from other Komi dialects [1]. The Izhma people, having adopted reindeer herding from the Nenets, developed it into a large-scale economy and became renowned traders throughout the Russian North. This specific way of life contributed to the formation of their unique local identity, and historically, even led to the emergence of the stereotype "cunning Izhma people" or "Jews of the North" [4]. Accordingly, the lexicon of Izhma dialect material culture not only reflects adaptation to the harsh conditions of the North but is also an important marker of the unique culture and historical continuity of the Izhma people. It demonstrates how linguistic contacts and traditional economic practices have shaped the dialect's vocabulary throughout history.

The relevance of comparing lexicographical data from the 19th and 21st centuries is due to several factors. Early Komi (Zyrian) language glossaries and phrasebooks, starting from the 18th century, already contained Russian lexical elements and even syntactic constructions, indicating a deep and prolonged history of language contacts [5]. Comparing vocabulary recorded in the 19th century with modern data allows tracing the dynamics of linguistic changes over almost two centuries. Such a diachronic analysis makes it possible to assess the degree of preservation of indigenous vocabulary against the background of active processes of borrowing, innovation, and archaization. Diachronic comparison also reveals how new realities and changes in the Izhma people's way of life are reflected in the language through borrowings. If old borrowings demonstrate deep phonetic and morphological adaptation, and new ones less so, this indicates an ongoing process of unilateral convergence, where the Russian language acts as a dominant source of influence. Such a phenomenon may also indicate a decrease in the ability or motivation of speakers to fully assimilate borrowings, which is a sign of increasing language shift [2]. Studies note the emergence of "linguistic nihilism" among young people, i.e., the loss of value of their native language and a preference for the dominant one [6]. Observing a significant replacement of indigenous terms with borrowings even in the basic vocabulary of material culture can be considered an alarming signal for the vitality of the language and calls into question the effectiveness of current measures for its preservation. Thus, comparing lexicographical data from different eras allows not only to record linguistic changes but also to gain a deeper understanding of the sociolinguistic processes affecting language viability.

Research Aim: To conduct a comparative analysis of the material culture lexicon of the Izhma dialect of the Komi language based on data from M.A. Castrén's dictionary (1844 / 2022) [7] and the audio dictionary of the Beloyarsk village dialect (2012) [8] to identify the main trends in lexical dynamics.

2. Research Material

M.A. Castrén's dictionary, titled *Elementa grammatices Syrjaenae* (1844) [9], represents the first published systematic overview of the Izhma dialect of the Komi language. Despite its Latin title, Castrén's work is primarily dedicated specifically to the Izhma dialect (and not to the Komi-Zyrian language as a whole), which makes it a unique lexicographical monument [10]. The materials for the dictionary were collected by M.A. Castrén during expeditions to Lapland and Northern Russia in 1838–1849, during which the scholar recorded data on almost thirty languages [11]. Castrén's dictionary was published in 1844, shortly after he defended his dissertation, and includes about 1100 lexical units, as well as a grammatical sketch. In addition, the book included samples of dialectal text – translations of four chapters of the Gospel of Matthew into the Izhma dialect. A critical edition of the dictionary, prepared under the editorship of P. Kokkonen and J. Rueter, was published in 2022 as

part of the *Manuscripta Castreniana* series [12]. Unlike the original Latin edition of 1844, the new version is an English-language annotated publication with detailed comments, corrections, and a translation of the grammatical description and dictionary. The 2022 edition clarifies the grapheme and transcription features of the original, includes data from manuscript and draft materials, which allows for a more accurate reconstruction of the Izhma dialect's features in the mid-19th century. Within the framework of this study, M.A. Castrén's dictionary (1844 / 2022) was uploaded to the LingvoDoc platform for subsequent lexicographical analysis.

One of the most comprehensive sources on the Ižma dialect is the audio dictionary of the Beloyarsk variety (Priural District, Yamal-Nenets Autonomous Okrug), compiled during a 2012–2013 field expedition led by E. V. Kashkin as part of the project Dialectological Atlas of the Uralic Languages of the Yamal-Nenets Autonomous Okrug (directed by N. B. Koshkareva). The dictionary contains 1,455 entries, each including a lexeme, phonological transcription, meaning, audio recording, paradigms with contextual usage, and cognates [8].

3. Description of the LingvoDoc Platform

The LingvoDoc platform is a software system designed for collaborative documentation and analysis of endangered languages. The system has been under development since 2012 and provides researchers with extensive functionality for linguistic research [13]. LingvoDoc allows the creation of multi-layered electronic dictionaries that can be linked to a geographical map, which is especially important for dialectological studies. The system provides flexible access control using Access Control Lists (ACLs), allowing groups of researchers to collaborate on dictionaries: editing, viewing, and publishing data. The platform supports various data types, including text, images, sound files (WAV, MP3, FLAC formats), as well as linguistic markup in ELAN and Praat formats. This flexibility is especially important for working with audio dictionaries, as it allows attaching original audio recordings to dictionary entries and conducting their detailed phonetic analysis.

LingvoDoc integrates built-in computational algorithms that significantly accelerate linguistic analysis compared to traditional methods. These include automated phonological calculations, duplicate detection and elimination (deduplication), and even preliminary etymological analysis of data. The use of LingvoDoc allows overcoming a number of limitations of traditional lexicography. Old printed dictionaries, especially those created before the advent of sound recording, often contain inaccuracies in transmitting the sound of words, which cannot be verified without audio materials. For example, in B. Munkácsi and B. Kálmán's Mansi dictionary (1986), there are 2–3 transcription variants for the same word form, and without audio data, it is difficult to establish which one corresponds to real pronunciation [14]. The LingvoDoc platform, due to the ability to attach audio files and process them (including with tools like Praat), allows for experimental-phonetic analysis and significantly increases the accuracy of phonetic transcription of words [13]. This not only improves the quality of the collected data but also opens up new opportunities for dialectological and etymological research, making them more objective and verifiable. For endangered languages, where there is little time left for documentation, such technology is invaluable.

4. Results

To identify patterns of lexical changes in the Izhma dialect of the Komi language, a comparative analysis of the material culture lexicon was conducted across five thematic groups: "Dwelling," "Utensils and Household Items," "Clothing and Footwear," "Tools and Crafts," and "Transport." This approach allowed tracing how lexical units transformed depending on their sphere of functioning and socio-cultural changes in the speakers' daily lives. The analysis was based on data from two lexicographical sources: M.A. Castrén's dictionary (1844), including 127 lexemes, and the audio dictionary of the Beloyarsk village dialect (2012), containing 167 lexemes. All lexemes were distributed into thematic groups and compared by the following parameters: presence/absence in the other source, phonological form, meaning, etymology. Within each group, the main types of lexical

transformations were recorded: direct correspondence, archaization, innovation, lexical replacement, phonetic-morphological change. The results of the linguistic and statistical analysis for each thematic group are presented below, accompanied by a precise rendering of phonetic forms from the sources and reliance on etymological data by [15].

4.1 Dwelling and Architecture

In the thematic group "Dwelling and Architecture" 23 lexemes represented in one or both dictionaries were analyzed (Table 1). 8 direct correspondences (30%) were recorded, in which lexemes are preserved with minor phonetic modifications: *kerka – k’e̮rka* 'house', *pyysjan – pɨs’an* 'sauna', *tagäs – tag’es* 'threshold', *veit – v’ejt* 'roof', *jör → jer* 'yard', etc. This indicates the relative stability of basic terms related to traditional constructions.

Table 1. Comparative analysis of the lexicon of the studied dictionaries by theme "Dwelling and Architecture".

No.	Meaning	Castrén 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
1.	bathhouse	pyysjan	pɨs'an	Pre-Permian *p8lɜ- ‘heat, steam’ (234)
2.	house	kerka	k'erka	Common Permian *ker + *k8ta (121–122)
3.	room	z'yr / žyr	ʒɨr	< Russian *жира* ‘dwelling’ (103)
4.	roof	veit / vejt	v'ejt	Pre-Permian *v8lɜ- ‘surface’ (49)
5.	window	ös'yn / öšyn	øk'en'tɕa, əʃɨn	Komi **öcінь* ‘opening’ < *öc-* ‘door’ + *-іньь* ‘diminutive suffix’ (213)
6.	threshold	tagäs	tag'es	Pre-Permian *t8ŋk8- ‘threshold’ (277)
7.	garden, yard, fence	jör	jer ‘enclosure’	Pre-Permian *jɜrɜ- ‘yard, enclosure’ (113)
8.	shed, small cattle yard	s'y / šy	ʃɨ	Common Permian *šo̮ ‘shed, premise’ (325)
9.	ceiling	jirt	-	? Mari *йорган* ‘blanket’ (111)
10.	barn, storage	kum	-	? Fin. *kumo* ‘threshing barnyard’, borrowed from Komi into North Russian, Khanty, Mansi (145)
11.	dwelling, shed	alacjug	-	< Old Rus. *алачуга* ‘hut’ < Turkic (389)
12.	cellar	-	pagrəm	Not in dictionary
13.	kitchen	-	kətɕás	Pre-Permian *ko̮čkɜ- ‘bend, curve, angle’ (138)
14.	furnace bottom	-	pod	< Indo-Iranian, cf. Sanskrit pā́d (223)
15.	subfloor	-	pos u	Common Permian *po̮s or *po̮s (226)
16.	ceiling beam	-	matɨtɕa	Not in dictionary
17.	corner	-	p'el'es, rətɕ	Formed from pel’ < Pre-Permian *pel’jä ‘ear’; Finno-Ugric *rät’s’ (218, 244)
18.	attic	-	vɨʃka	Not in dictionary
19.	birch bark covers for summer tents	-	jodum	Not in dictionary
20.	poles in chum for hanging hooks over fire	-	tsɨnzɨ	< Nenets *сымзы* ‘vertical pole inside chum’ (268)
21.	chum cover	-	n'uk	Not in dictionary
22.	mat of twigs in chum	-	n'eru	Common Permian *n’o̮r ‘vine, branch, twig’ (198)
23.	inner chum cover	-	pɐd'n'itɕtɕe	Not in dictionary

A significant portion of lexemes — 12 units (49%) — refers to innovations, presented in the 2012 audio dictionary but absent in Castrén's. These include, in particular, terms describing elements of a chum: *tsɨnzɨ* 'poles in a chum', *jodum* 'birch bark flooring', *n'eru* 'woven mat', *n'uk* 'chum covering', *pɐd'n'itɕtɕe* 'inner covering of a chum', etc. These elements are associated with a nomadic lifestyle and were likely absent in Castrén's area of fixation. Archaic terms (lexemes recorded by Castrén but not present in the audio dictionary) include 3 units (13%): *kum* 'barn', *jirt* 'ceiling', *alacjug* 'dwelling, shack'. The latter word goes back to Old Russian *алачуга* < Turkic, and likely fell out of active use. Thus, the lexicon of dwelling in the Izhma dialect demonstrates a balance between the preservation of stable basic terms and the expansion of the semantic field due to innovative and borrowed vocabulary, primarily related to mobile northern life. A tendency towards active vocabulary replenishment through interaction with Nenets and Russian languages is observed.

4.2 Utensils and Household Items

The materials of the thematic group "Utensils and Household Items" reflect both the stability of the basic vocabulary of the Izhma dialect and its development in conditions of changing cultural and household realities (Table 2). A total of 67 lexemes were involved in the comparison, recorded either in M.A. Castrén's dictionary (1844), or in the audio dictionary of the Beloyarsk village dialect (2012), or in both. The largest share consists of lexemes represented only in the 2012 dictionary – 32 units (47.8%). Many of them reflect either new household items characteristic of the 20th–21st centuries, or regional specifics not captured in Castrén's expeditionary dictionary. These include, for example, *katɕaj* 'swing', *p'el'en'ítɕa* 'diaper', *jodum* 'birch bark covers for summer chums', *n'ebl'uj* 'young reindeer hide', *balagán* 'curtain'. Some units go back to Nenets or Russian borrowings, others continue the indigenous vocabulary of the Izhma dialect (*kɨʃ* 'case', *n'eru* 'mat').

Table 2. Comparative analysis of the lexicon of the studied dictionaries by theme "Utensils and Household Items".

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
1.	doll	akanj	akan'	Based on Komi language, -ань – diminutive suffix (31)
2.	spoon, ladle	dar	dar	Common Permian *dar8 < Iranian, cf. Sanskrit darvi- 'spoon' (87)
3.	birch bark vessel	dooz	dɔz	Common Permian *dǫz or *döz 'type of dish' (94)
4.	mortar	gyr	gɨr	Pre-Permian *kūre- 'hollowed wooden vessel' (85)
5.	needle	jem	jem	Proto-Uralic *äjmä (99)
6.	basket, box (for storing dishes in a chum)	kuda	kuda	Pre-Permian *k8nt3 'bag' (411)
7.	pot, vase	gyrnic / gyrnjicj	-	< Old Russian *гърньць* 'pot' (86)
8.	matches	iiztäg	-	Komi *iz-teg 'solid combustible sulfur' (110)
9.	pillow	juruu	-	Pre-Permian *jūre 'base', 'root', 'head' (335) + Pre-Permian *ala- 'low' (295)
10.	bowl	kumka	-	Not in dictionary
11.	bath broom	koräs / koräsj	-	Derived from *кор* 'leaf' (134). Pre-Permian *kor- 'leaf' (133).
12.	hand spindle	kozjalj	-	Not in dictionary
13.	clay pot, vessel	kub	-	Not in dictionary

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
14.	stool, footrest	kokuu	-	Common Permian *kǫk ‘leg, foot’ (129) + Pre-Permian *ala- ‘low’ (295)
15.	bandage, dressing	körtäd	-	*кӧрт- (кӧртавны)* + derivational suffix *-öд* (142). Pre-Permian *kert3- ‘to tie, to bind, to fasten’ (142)
16.	feeder, feeding trough	ljasni / ljasjnji	-	< Russian *ясли* (393)
17.	broom; spruce branches	lys	lɨs	Pre-Permian *lüs ‘conifer needle’ (164)
18.	whip	ors	ors	Common Permian *ors ‘whip’ (209)
19.	knife	purt	purt	< Iranian, cf. Old Persian *para ðu- ‘knife’ (233)
20.	needle eye	pys	pɨs	Common Permian *pùs-. Pre-Permian history of the word is unclear (237)
21.	hook	vugyr	vugúr wugúr	Pre-Permian *waŋkɜ- ‘hook, handle’ < Indo-Iranian, cf. Sanskrit vaŋka- ‘curved, bent’ (69)
22.	scissors	s’yran / šyran	ʃɨran	Pre-Permian * šɜrɜ- ‘to cut, to slice’ (327)
23.	kettle	pört	-	Pre-Permian *pɞrttɜ ‘dish, vessel’ (229)
24.	cauldron	-	pərt	Pre-Permian *pɞrttɜ ‘dish, vessel’ (229)
25.	towel	suläk	-	< Russian *сулок* ‘small width, small towel’ (266);
26.	towel	-	kɨt'iʃkán	Derived from *чышкыны* ‘to wipe off, to wipe clean’. Pre-Permian *č8čɜ (315)
27.	pole, post	majeg	-	Common Permian *majɜg < Iranian; cf. Ossetian mēχ, mīχ ‘stake, stick’ (168)
28.	rag	rusum	-	The root of the word is *руз,* which is also seen in words *руз-раз* ‘scattered’, *руз-размунны* ‘to crackle, to collapse, to disperse’, *рузйыны* ‘to wear out (clothes, shoes)’ (245);
29.	rag	-	bon	< Iranian, cf. Old Indian *bandhá* (40)
30.	shingle, lath	sartas	-	Pre-Permian *sărta- ‘chip’ (250)
31.	candle	sis / sjisj	s'is'	Pre-Permian *s’ištɜ or s’ikštɜ ‘wax’ (257)
32.	salt shaker	solantäg	-	< Veps-Karelian *solan-tohi or *solan-tuohi ‘birch bark (for) salt’ (261)
33.	comb	synan	-	Common Permian *sùŋ- (268)
34.	plate, cup	tasti / tasjti	-	< Iranian, cf. Old Persian taš’ti ‘cup’ (278)
35.	ointment, salve	maitas / majtas	-	Common Permian *maj- ‘to smear, to crumble finely’ (167)
36.	nail, peg, plug	tuu	-	Pre-Permian *tola (284)
37.	broom, brush	tug	-	Pre-Permian *t8̈nkä ‘end, top, protruding part’ (285)
38.	lining	uutys	-	*ув* ‘bottom’, *увт* ‘padding’ (295)
39.	mirror	vizlysjan / vizjlysjan	-	Common Permian *vɜʒ’- ‘to guard, to hold, to contain, to look’; Pre-Permian *wičɜ- ‘to guard, to be careful’ (55-56)
40.	bed	volj	-	*Воль* ‘hide’ + *пась* ‘clothing’. Pre-Permian *oð’e- ‘bed’ (62)
41.	insole	-	vol'es	Pre-Permian *oð’e- ‘bed’ (62)

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
42.	pole, perch	zib	-	Not in dictionary
43.	perch, pole	zor	-	Not in dictionary
44.	flint	bija iiz	-	No generally accepted etymology (40)
45.	hide for sitting on a sled	-	amd'ór	Not in dictionary
46.	handle (of bucket, etc.); knob (of door, etc.)	-	vug wug	Pre-Permian *waŋkɜ- 'hook, handle' < Indo-Iranian, cf. Sanskrit vaŋka- 'curved, bent' (69)
47.	birch bark box	-	b'erójʃka b'irójʃka	Not in dictionary
48.	bolt, latch	-	igan	Common Permian *jùg- 'latch, bolt' (109)
49.	birch bark covers for summer chums	-	jodum	Not in dictionary
50.	broom (for sweeping)	-	gəl'ik	Not in dictionary
51.	swing	-	katɕaj	Not in dictionary
52.	stove damper	-	juʃka	Not in dictionary
53.	young reindeer hide (up to 1 year old)	-	n'ebl'uj	< Nenets *няблюй* 'calf hide, removed approximately in September, suitable for malitsa' (187)
54.	shoe stretcher	-	kəpɨl	Not in dictionary
55.	diaper, swaddling cloth	-	p'el'en'íɕa	Not in dictionary
56.	dish box	-	p'agr'ev'itɕ	Not in dictionary
57.	washstand, hand-washer	-	rukəm'éj	Not in dictionary
58.	reindeer hide bag for sewing accessories	-	tutɕu	< Nenets туця'(н) '(small) bag for sewing accessories' (288)
59.	thread spool	-	turetɕka	Not in dictionary
60.	box; sled body	-	lar	Not in dictionary
61.	mat in a chum	-	kəv'er	Not in dictionary
62.	grass mat	-	nor'i	Not in dictionary
63.	twig mat in a chum	-	n'eru	Common Permian *n'o̯r 'vine, branch, twig' (198)
64.	blanket	-	əd'jála	Not in dictionary
65.	canopy, curtain	-	balagán	Not in dictionary
66.	thimble	-	ɕeun'kúmku	Pre-Permian *č'ʊ̈ŋe- 'finger' (314)
67.	cover, case	-	kɨʃ	Common Permian *kùš 'peel, shell' (155)

Direct correspondences between sources account for 16 lexemes (23.9%). This category includes, in particular: *dar* 'spoon', *dooz – dɔz* 'box', *gyrnic – gɨr'ič* 'pot', *gyr – gɨr* 'mortar', *jem* 'needle', *kuda* 'basket', *lys – lɨs* 'broom', *ors* 'whip', *purt* 'knife', *pys – pɨs* 'eye of a needle', *šyran – ʃɨran* 'scissors', *sisj – s'is'* 'candle', *vugyr – vugúr / wugúr* 'hook'. The last example is of particular interest: the audio

dictionary presents forms with *v* and *w*, while Castrén's dictionary (2022) only records the form with *v*. Given the editorial normalization of spelling in the 2022 publication (replacement of *w* with *v*), it can be assumed that the original Castrén recording used the form *wugyr*, which is preserved in modern speech.

Archaic terms, i.e., lexemes recorded only in Castrén's dictionary and absent in the 2012 dictionary, comprise 16 lexemes (23.9%). Among them are: *kozjalj* 'spinning wheel', *ljasni* 'feeding trough', *solantäg* 'salt shaker', *tasti* 'plate', *vizlysjan* 'mirror', *uutys* 'gasket', *sartas* 'shingle', *synan* 'comb'. Some of these lexemes may have fallen out of active use, while others may not have been recorded in the audio collection for various reasons. Semantic discrepancies (shifts) were noted in 1 case (1.5%) — the pair *pört* (in Castrén's dictionary: 'kettle') and *pərt* (in the 2012 dictionary: 'pot'). Despite the coincidence of the phonological form, the denoted objects differ in purpose, which may indicate a reinterpretation of the lexeme in the context of a changing material world. Lexical replacements were recorded in 2 cases (3.0%): *suläk* 'towel' (1844) → *kít'iʃkán* (2012), *rusum* 'rag' (1844) → *bon* (2012). However, even here, it cannot be excluded that Castrén's original lexemes were simply not recorded in the dictionary.

4.3 Clothing, Footwear, and Adornments

The lexicon denoting clothing, footwear, and decorative elements demonstrates a stable connection with tradition, but at the same time reveals pronounced innovative processes (Table 3). The analysis table included 28 lexemes, compared based on M.A. Castrén's dictionary (1844) and the 2012 audio dictionary of the Beloyarsk village dialect. Innovations – lexemes present only in the 2012 dictionary – account for 11 units (39,3 %). Among them are borrowed forms: *p'im* 'pimy', *sap'eg* 'boot', *t'øbek* 'tobors', *gatɕ* ' trousers', *pandi* 'hem trim of malitsa', as well as indigenous lexemes absent in Castrén: *ʃir* 'shoelace for clothing ', *von* 'shoelace', *kɨz'* 'button', *sir* 'fringe', *tɕiʃjan / tɕuʃ'jan* 'kerchief'.

Table 3. Comparative analysis of the lexicon of the studied dictionaries by theme "Clothing, Footwear, and Adornments".

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
1.	belt	jy / ji	ji	Pre-Permian *jäje 'belt' (111)
2.	ring	kyc'j / kyčj	t'un' kɨtʃ	Pre-Permian *kɨ̈ðcä 'circle' (155); Pre-Permian *č'ðŋe- 'finger' (314)
3.	footwear	kömkot	kəmkot	Compound word: *köм* 'footwear' and кот 'burial footwear', in the past also 'footwear'; during the formation of this compound word, *köм* and *кот* probably designated two varieties of footwear (141)
4.	fur gloves	kypys / kepysj	køpɨs'	*Ке-пысь* 'hand'-'mitten'. Pre-Permian *pɜs'ɜ 'mittens' (121)
5.	fabric, cloth	noj	noj	Origin is unclear (193)
6.	parka	parka	parka	Not in dictionary
7.	sleeve	sos	sos	Derived from *сой* 'hand' with denominal suffix *-с*. Common Permian *sǫjɜs(к-) (262)
8.	shirt	dörem	-	Common Permian *dɛrem 'shirt' (-m is a suffix) (96).
9.	headband for the bride	jurnoj	-	Pre-Permian *jūre 'base', 'root', 'head' (335) + Common Komi nǫj 'cloth'. Origin is unclear (193)
10.	linen clothing, canvas	döra	-	Common Permian *dɛra 'fabric' (96)
11.	pair of shoes	köm	-	Pre-Permian *kɨ̈me- 'footwear' (141)

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
12.	bandage, dressing	körtäd	-	From *көрт- (көртавны)* + derivational suffix *-öд* (142). Pre-Permian *kertɜ- 'to tie, to bind, to fasten' (142)
13.	women's cloak, fur coat (in general)	pas / pasj	-	Common Permian *pas' 'fur coat' (217)
14.	clothing	paskam / pasjkäm	-	Common Permian *pas' 'fur coat' (217) + *көм* 'footwear'
15.	pocket	zep / zjep	-	Wichmann assumes borrowing from Old Chuvash or from Russian with initial z' (105)
16.	rag, cloth	-	dəra / dərá	Common Permian *dɛra 'fabric' (96)
17.	scarf	-	tɕiʃjan / tɕuʃ'jan	Derived from *чышкыны* 'to wipe off, to wipe clean'. Pre-Permian * č8čɜ (315)
18.	button	-	kɨz'	Common Permian *kŭ̆ʒ- 'piece of fabric for joining edges of clothing (footwear) by tying a knot' (123)
19.	stockings	-	l'ip'itɕa	Not in dictionary
20.	boot	-	sap'eg	Not in dictionary
21.	shoelace	-	von'	Pre-Permian *wüŋä or *üwä 'belt, sash' (67)
22.	pimy (traditional Komi winter boots)	-	p'im	< Nenets p'iemmp 'reindeer fur footwear' (221)
23.	cord, lace	-	ʃɨr	Common Permian *šu̇r- (326)
24.	fringe	-	sɨr	Common Permian *su̇r- (268)
25.	tobory (type of footwear)	-	t'øbek	< Samoyedic, cf. Yurak tōBrā 'reindeer hide footwear' (283)
26.	fabric	-	təvar	Not in dictionary
27.	trousers	-	gatɕ	< Russian гачи 'trousers' (75)
28.	trim of the malitsa hem	-	pandɨ	< Nenets, cf. pāntat, панд 'trim, patch (on the hem)' (216)

Direct correspondences were established for 7 lexemes (25.0%): *kypys [kepysj] – køpɨs'* 'mittens', *noj* 'fabric', *sos* 'sleeve', *parka* 'parka', *jy [ji]* 'belt', *dörem – dɤrem* 'shirt', *kömkot – kəmkot* 'footwear'. These lexemes demonstrate the stability of the basic corpus of traditional clothing and only minor phonetic discrepancies. Archaic terms, present only in Castrén's dictionary and absent in the 2012 audio dictionary, comprise 8 units (28.6%): *paskam [pasjkäm]* 'clothing', *pasj* 'women's cloak, fur coat (in general)', *köm* 'pair of shoes', *körtäd* 'bandage', *jurnoj* 'bride's headdress', *döra* 'cloth', *zep [zjep]* 'pocket'. These words may have fallen out of active use, been displaced by new terms, or not been captured in the audio recording of modern dialectal material. Lexical replacement was established in 1 case (3.6%): *kyčj → t'un' kɨtʃ* 'ring' — here a transition from a single-component designation to a descriptive construction is observed. Semantic discrepancies were recorded in 1 case (3.6%): *döra* (1844) 'cloth' → *dəra* (2012) 'rag' (possible shift in meaning).

4.4 Tools and Crafts

Thematic group shown in Table 4 includes 38 lexemes related to the economic and craft realities of the Izhma Komi. Comparing data from M.A. Castrén's dictionary (1844) and the audio dictionary of

the Beloyarsk village dialect (2012) allows identifying both stable elements of traditional terminology and tendencies of its loss or renewal.

Table 4. Comparative analysis of the lexicon of the studied dictionaries by theme "Tools and Crafts".

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
1.	stick, staff	bed	b'ed	< Pre-Permian *pɜntɜ (38)
2.	whetstone	zud	zud	Common Permian *z8̈d ‘whetstone’ (106)
3.	shovel	zyr	zɨr	Not in dictionary
4.	axe	cjer	ʦer	Common Komi *č’e̮r̄ ‘axe’ (303)
5.	knife	purt	purt	< Iranian, cf. Old Persian *para δu- ‘knife’ (233)
6.	sickle	cjarla	ʦarla	< Chuvash, cf. s’urla, s’orla ‘sickle’ (Turkic *čarla) (302)
7.	pitchfork	läbin	lebun	Derived from *лэптыны* (165), Common Permian *leb- ‘to fly’, *lɜb- ‘to float up’ (165)
8.	hook	vugyr	vugúr / wugúr	Pre-Permian *waŋkɜ- ‘hook, handle’ < Indo-Iranian, cf. Sanskrit vaŋka- ‘curved, bent’ (69)
9.	whip	ors	ors	< Iranian, cf. Sanskrit astra- ‘whip, scourge’ (208)
10.	dragnet	tyy	tɨ	Parallels with Khanty and Mansi (292)
11.	harrow	agas	-	< Baltic-Finnic *ägäs or *äƳäs (30)
12.	shaft, handle	voz’ / vož	-	Common Permian *wo̮ž- ‘shaft’. Pre-Permian *8č8- (60)
13.	ard (plowshare)	amäs / amäsj	-	< Common Permian *amɜž’ ‘plowshare’ < Iranian, cf. New Persian āmāž ‘plow handle’ (32)
14.	spear	s’i / ši	-	Pre-Permian *šuje, * šoje ‘spear’, ‘point’ (325)
15.	broom, brush	tug	-	Pre-Permian *t8̈nkä ‘end, top, protruding part’ (285)
16.	arc, bow	vudz' / vudž	-	Pre-Permian *wanče- ‘bow, fiddle bow’ (69)
17.	pole, perch	zib	-	Pre-Permian *sɜmpɜ (105)
18.	perch, pole	zor	-	Common Permian *zo̮r ‘small pole, lever’
19.	beam, pole for carrying water, yoke	sjoor	-	Common Permian *s’o̮ri (253)
20.	trap	läcj	-	Pre-Permian *läms̈ʊ̈ ‘snare’ (165)
21.	broom; spruce branches	lys	-	Pre-Permian *lüs ‘conifer needle’ (164)
22.	plow	gör	-	Pre-Permian *kɜrɜ- ‘to dig, to excavate, to pick’ (80)
23.	pole, post	majeg	-	Common Permian *majɜg < Iranian, cf. Ossetian mēχ, mīχ ‘stake, stick’ (168)
24.	wedge, key (carpenter's)	narvi	-	Etymology is unclear (186)
25.	butt of an axe	-	ʦer tɨʃ	Pre-Permian *t8čka- ‘butt (of an axe, hammer)’ (293)
26.	float	-	tab	Pre-Permian *t8mbɜ- (277)

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
27.	barrel	-	botɕka	Not in dictionary
28.	handle (of a bucket, etc.); handle (of a door, etc.)	-	vug / wug	Pre-Permian *waŋkɜ- 'hook, handle' < Indo-Iranian, cf. Sanskrit vaŋka- 'curved, bent' (69)
29.	sinker	-	k'eb'ed	Baltic-Finnic *kebed 'light' (119)
30.	thick short stick	-	pal'itɕ	< Russian (215)
31.	drill	-	napárja	Not in dictionary
32.	sawhorse (for cutting firewood)	-	kəzla	Not in dictionary
33.	shoe tree (for stretching shoes)	-	kəpɨl	Not in dictionary
34.	rifle	-	p'ɨʃʃál'	Not in dictionary
35.	wad (in a hunting rifle)	-	pl'as	Not in dictionary
36.	stern oar	-	pelis	Common Permian *pe̮lis 'oar' (218)
37.	hammer	-	mələt	Not in dictionary
38.	ryuzha (fishing gear)	-	r'uʒa	Not in dictionary

Direct correspondences were established for 7 lexemes (18.4%): *bed – b'ed* 'stick', *zud – zud* 'whetstone', *purt* 'knife', *cjer – ɕer* 'axe', *ors* 'whip', *läbin – lebu̮n* 'fork', *cjarla – ɕarla* 'sickle'. These cases demonstrate both the preservation of meanings and the closeness of phonological structure.

Archaic terms, present only in the 1844 dictionary, comprise 16 lexemes (42.1%): *agas* 'harrow', *amäs / amäsj* 'ard', *voz' / vož* 'handle', *s'i / ši* 'spear', *tug* 'broom', *tyy* 'seine', *vudz' / vudž* 'bow', *zib* 'pole', *zor* 'pole', *sjoor* 'yoke', *lys* 'brush made of coniferous branches', *gör* 'plow', *majeg* 'post', *narvi* 'wedge', *zyr* 'shovel', *läcj* 'trap'. Their absence in the 2012 dictionary can be explained both by the disappearance of the denoted realities and by the non-recording of these lexemes within the expeditionary sample. It is also possible that some words are preserved in the passive vocabulary of speakers but were not actualized in speech.

Innovations, represented only in the 2012 dictionary, number 14 lexemes (36.9%): *tab* 'float', *botɕka* 'barrel', *vug / wug* 'handle (of a bucket, door)', *k'eb'ed* 'sinker', *pal'itɕ* 'stick', *napárja* 'auger', *kəzla* 'trestle', *kəpɨl* 'shoe stretcher', *r'uʒa* 'fishing device', etc. It is probable that some of the lexemes presented only in the 2012 audio dictionary were also known in the 19th century, but for various reasons were not recorded by M.A. Castrén.

4.5 Transport and Harness Elements

The analysis of vocabulary related to transport and harness elements covers 23 lexemes (Table 5). A comparison of data from M.A. Castrén's dictionary (1844) and the audio dictionary of the Beloyarsk village dialect (2012) allows recording both elements of stable nomination and signs of terminological renewal.

Direct correspondences were established for 4 lexemes (17.4%): *pyž – pɨʒ* 'boat', *dadj – dod'* 'sledge', *pur* 'raft', *megyr – m'egɨr* 'bow (horse harness)'. These lexemes retain not only their meaning but also their structural-phonetic features, reflecting the stability of basic transport terminology.

Archaic terms, recorded only in the 1844 dictionary, are represented by 2 lexemes (8.7%): *söpecj* 'rudder', *sijes [sjijes]* 'collar (part of harness)'. Their absence in the 2012 dictionary may be associated with the loss of the denoted realities or with the specifics of field recording, where these units were not actualized in the informants' speech.

Table 5. Comparative analysis of the lexicon of the studied dictionaries by theme "Transport and Harness Elements".

No.	Meaning	Castrén, 1844 / 2022	Beloyarsk Dictionary, 2012	Lytkin & Gulyaev, 1999
1.	boat	pyz' / pyž	pɨʒ	Pre-Permian *pučɜ (235)
2.	raft	pur	pur	Pre-Permian *p8rɜ, *parɜ 'heap', 'crowd', 'flock' (233)
3.	sleigh	dadj	dod'	Common Permian *dǫ̈d'ɜ (94)
4.	horse collar	megyr	m'egɨr	Common Permian *meg 'bend, curve' (171)
5.	rudder	söpecj	-	< Baltic-Finnic via Russian. Finnish *sapsi, sapsa, sapso* 'rudder, helm' > Russian (Siberian and archaic) *сопец* 'rudder, rule' (263)
6.	collar (part of horse harness)	sijes / sjijes	-	Common Permian *s'ijes (255)
7.	hide for sitting on a sleigh	-	amd'ór	Not in dictionary
8.	strap	-	l'amka	Not in dictionary
9.	stern oar	-	pelɨs	Common Permian *pe̮lis 'oar' (218)
10.	sleigh for transporting good clothing, soft tanned hides	-	van'd'éj	Not in dictionary
11.	rattle (on a reindeer)	-	tatɛkan	Not in dictionary
12.	box; sleigh body	-	lar	Not in dictionary
13.	bone piece or iron ring for holding the rein	-	xalsúla	Not in dictionary
14.	pole used to drive reindeer in harness	-	xar'éj	Not in dictionary
15.	reindeer harness	-	sal'ámka	< Nenets *ca* 'trace' + < Russian *лямка* 'strap' (248)
16.	button in reindeer harness	-	tɛalak	Not in dictionary
17.	women's sleigh (for a woman with a child)	-	bəlka dad'	Not in dictionary
18.	bridle	-	s'erm'ed	Pre-Permian *s'erm8- (251)
19.	hunting skis	-	l'ampa	< Nenets lampɐ (166)
20.	skis	-	lɨz'	> Old Russian *лыжь, лыжи* 'skis' (164)
21.	box; sleigh body	-	lar	Not in dictionary
22.	rowing oar	-	øpásnej	Not in dictionary
23.	place for a child in women's sleigh	-	bəlk	Not in dictionary

Innovations, exclusively present in the audio dictionary, comprise 17 lexemes (73.9%). These include both borrowed units (*sal'ámka, lar, bəlka dad'*) and lexemes denoting Arctic transport realities (*xar'éj* 'driving pole', *l'ampa* 'hunting skis', etc.). The predominance of the innovative layer indicates the active development of the thematic subsystem under the influence of ecological, economic-cultural, and interlingual factors. Besides the objective change of realities, a probabilistic factor cannot be excluded: some lexemes might have functioned in the 19th century as well, but were not recorded by

M.A. Castrén due to limited contact time with informants and the peculiarities of the dialectal situation.

5. Conclusions

The conducted comparison of the material culture lexicon in M.A. Castrén's dictionary (1844) and the audio dictionary of the Beloyarsk village dialect, Priuralsky District, Yamalo-Nenets Autonomous Okrug (2012), revealed a wide range of lexical correspondences and discrepancies, conditioned both by the temporal gap between the sources and by different methodological approaches to recording linguistic material. The analysis covered 5 thematic groups (Table 6): dwelling and architecture, utensils and household items, clothing, footwear, and adornments, tools and crafts, and transport. In total, 183 lexemes were examined: 127 from Castrén's dictionary and 167 from the 2012 dictionary. This allowed identifying quantitative and qualitative patterns in the development of the Izhma dialect lexicon of the Komi language over more than 160 years.

The largest proportion of archaisms (lexemes present only in the 1844 dictionary) was recorded in the "Tools and crafts" group (50%), which likely reflects the disappearance of the denoted objects due to the loss of traditional crafts. At the same time, the maximum number of innovations (lexemes present only in the 2012 dictionary) was found in the "Transport" group (73.9%), which is explained by borrowings and the active development of new Arctic terminology. The study also showed that not all differences between the dictionaries should be interpreted as a result of lexical changes: the absence of a number of lexemes in one of the sources may be related to incomplete recording, peculiarities of field conditions, or a focus on different thematic layers. Thus, the lexicon of material culture in the Izhma dialect of the Komi language demonstrates both areas of stability and active lexical restructuring. The obtained data can be used for reconstructing lexical subsystems, studying contact influences, and describing linguistic dynamics on the periphery of the Komi language area.

Table 6. Distribution of Lexemes by Semantic Groups and Types of Correspondence.

Semantic Group	Total Lexemes	Direct Equivalents	Lexical Replacements	Archaisms	Innovations
Dwelling and Architecture	23	6 (26.1%)	2 (8.7%)	2 (8.7%)	12 (52.2%)
Utensils and Household Items	67	17 (25.4%)	2 (3%)	18 (26.9%)	30 (44.8%)
Clothing, Footwear, and Adornments	28	7 (25%)	2 (7.1%)	8 (28.6%)	7 (25%)
Tools and Crafts	38	7 (18.4%)	1 (2.6%)	19 (50%)	11 (28.9%)
Transport and Harness Elements	23	4 (17.4%)	0 (0%)	2 (8.7%)	17 (73.9%)

References

[1]. M. A. Sakharova, N. N. Selkov, "Izhma Dialect of the Komi Language". Syktyvkar: Komi Branch of the USSR Academy of Sciences, Institute of Language, Literature and History, 1976.

[2]. M. Leinonen, "Russian influence on the Iźma Komi dialect", International Journal of Bilingualism, vol. 13, no. 3, pp. 309–329, 2009.

[3]. V. Denisenko, T. Klyanova, "Problems of the Native Language among the Komi Rural Population", Orientir: Socio-Political Journal of the Komi Republic, vol. 2, no. 1-2, pp. 48–50, 1994.

[4]. O. V. Kotov, M. B. Rogachev, Yu. P. Shabaev, "Modern Komi People". Ekaterinburg: Ural Branch of the Russian Academy of Sciences, 1996.

[5]. A. S. Sidorov, “Influence of the Russian Language on the Grammatical Structure of the Komi Language”, In: G. V. Fedyuneva (Ed.), Selected Articles on the Komi Language. Syktyvkar, 1992, pp. 102–120.
[6]. N. M. Bichurina, “Mountains, Language and a Bit of Social Magic: An Experience in Critical Sociolinguistics”. Saint Petersburg: European University at St. Petersburg Publishing House, 2021.
[7]. Dictionary of the Izhma Dialect of the Komi Language Based on the Materials of M.A. Castrén (1843–1844), According to the Elementa grammatices Syrjaenae (1844), Ed. by Paula Kokkonen and Jack Rueter (Syrjaenica, 2022) Available at: https://lingvodoc.ispras.ru/dictionary/11470/1/perspective/11470/2/view (accessed: July 3, 2025).
[8]. Dictionary of the Izhma dialect of the Komi-Zyrian language, Beloyarsk village dialect of Priuralsky District, Yamal-Nenets Autonomous Okrug, Ed. by E. V. Kashkin, 2012. Available at: https://lingvodoc.ispras.ru/dictionary/1389/6/perspective/1389/7/view (accessed: July 3, 2025).
[9]. M. A. Castrén, “Elementa grammatices Syrjaenae”. Saint Petersburg, 1844.
[10]. J. Janhunen, “Manuscripta Castreniana: A General Preface to the Series”, Manuscripta Castreniana. Finno-Ugrian Society, 2017.
[11]. N. Partanen, M. Hämäläinen, J. Rueter, K. Alnajjar, “Processing M.A. Castrén's Materials: Multilingual Typed and Handwritten Manuscripts”, 2021. Available at: https://arxiv.org/abs/2112.14153 (accessed: July 4, 2025).
[12]. M. A. Castrén, “Elementa grammatices Syrjaenae. Critical edition with introduction, translation and commentary”, Ed. by Paula Kokkonen, Jack Rueter. Helsinki: Finno-Ugrian Society, 2022. 278 p. (Manuscripta Castreniana; Vol. VI).
[13]. Yu. V. Normanskaja, O. D. Borisenko, I. B. Beloborodova, A. I. Avetisyan, “The Software System LingvoDoc and the Possibilities It Offers for Documentation and Analysis of Ob-Ugric Languages”, Doklady Mathematics, vol. 105, no. 3, pp. 187–206, 2022.
[14]. B. Munkácsi, B. Kálmán, “Wogulisches Wörterbuch”. Budapest: Akadémiai Kiadó, 1986.
[15]. V. I. Lytkin, E. S. Gulyaev, “Brief Etymological Dictionary of the Komi Language”. Syktyvkar: Komi Book Publishing House, 1999. 430 p.

Информация об авторе / Information about author

Ольга Николаевна БАЖЕНОВА – кандидат филологических наук, научный сотрудник лаборатории Лингвистических платформ Института системного программирования им. В.П. Иванникова РАН. Сфера научных интересов: диалектология, сравнительно-историческое языкознание, история пермских литературных языков.

Olga Nikolayevna BAZHENOVA – Cand. Sci. (Philology), researcher of the Laboratory of Linguistic Platforms, Ivannikov Institute for System Programming of the Russian Academy of Sciences. Her research interests include dialectology, comparative-historical linguistics; history of the Permic literary languages.

DOI: 10.15514/ISPRAS-2025-37(6)-14

Сегментация документов на основе графовых нейронных сетей: от строк к словам

[1,3] *Д.Е. Копылов, ORCID: 0009-0000-6348-4004 <it-daniil@yandex.ru>*
[1,2] *А.А. Михайлов, ORCID: 0000-0003-4057-4511 <mikhailov@icc.ru>*
[1,3] *Р.И. Трифонов, ORCID: 0009-0006-0024-8964 <tr1fonov.roman@yandex.ru>*

[1] *Институт динамики систем и теории управления имени В.М. Матросова СО РАН, Россия, 664033, г. Иркутск, ул. Лермонтова, д. 134.*

[2] *Институт системного программирования РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[3] *Институт математики и информационных технологий Иркутского государственного университета, Россия, 664003, Иркутск, бульвар Гагарина, д. 20.*

Аннотация. В работе представлен метод анализа макета PDF документов на основе графовых нейронных сетей (GNN), использующий слова в качестве узлов графа для преодоления ограничений современных подходов, опирающихся на строки или локальные области. Предложенная модель WordGLAM, основанная на модифицированных графовых сверточных слоях, демонстрирует возможность построения иерархических структур через агрегацию слов, что обеспечивает баланс между точностью детекции элементов и их семантической связностью. Несмотря на отставание от лидирующих моделей в данной области (например, от модели Vision Grid Transformer) по метрикам точности, исследование выявляет системные проблемы области: дисбаланс данных, неоднозначность кластеризации слов («цепные связи», «мосты» между несвязанными регионами), а также спорные критерии выбора классов при разметке. Ключевым вкладом работы является формулировка новых исследовательских задач, включая оптимизацию векторных представлений слов, учет признаков ребер и разработку методов оценки для сложных иерархий. Результаты подтверждают перспективность подхода для создания адаптируемых моделей, способных обрабатывать разноформатные документы (научные статьи, юридические тексты). Работа фокусирует внимание на необходимости дальнейших исследований в области регуляризации и расширения обучающих данных, открывая пути для улучшения переносимости методов анализа макета на новые домены. Код и модели были опубликованы на GitHub (https://github.com/YRL-AIDA/wordGLAM).

Ключевые слова: графовые нейронные сети; сверточные графовые нейронные сети; сегментация документа; анализ макета документа; регионы документа; блоки документа; сегменты документа.

Для цитирования: Копылов Д.Е., Михайлов А.А., Трифонов Р.И. Сегментация документов на основе графовых нейронных сетей: от строк к словам. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 219–232. DOI: 10.15514/ISPRAS–2025–37(6)–14.

Благодарности: Работа выполнена в рамках государственного задания Министерства науки и высшего образования Российской Федерации (тема № 1023110300006-9).

Segmentation of Documents Based on Graph Neural Networks: from Strings to Words

[1, 3] *D.E. Kopylov, ORCID: 0009-0000-6348-4004 <it-daniil@yandex.ru>*
[1, 2] *A.A. Mikhailov ORCID: 0000-0003-4057-4511 <mikhailov@icc.ru>*
[1, 3] *R.I. Trifonov, ORCID: 0009-0006-0024-8964 <tr1fonov.roman@yandex.ru>*

[1] *Matrosov Institute for System Dynamics and Control Theory of the Russian Academy of Sciences, 134, Lermontov st., Irkutsk, 664033, Russia.*

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

[3] *Irkutsk State University Institute of Mathematics and Information Technologies, 20, Gagarin Boulevard, Irkutsk, 664003, Russia.*

Abstract. The paper presents a method for analyzing the layout of PDF documents based on graph neural networks (GNN), which uses words as graph nodes to overcome the limitations of modern approaches based on strings or local areas. The proposed WordGLAM model, based on modified graph convolutional layers, demonstrates the possibility of constructing hierarchical structures through word aggregation, which ensures a balance between the accuracy of element detection and their semantic connectivity. Despite lagging behind state-of-the-art models (for example, Vision Grid Transformer) in accuracy metrics, the study reveals systemic problems of the region: data imbalance, ambiguity in word clustering ("chain links", "bridges" between unrelated regions), as well as controversial criteria selecting classes in the markup. The key contribution of this work is the formulation of new research tasks, including optimization of vector representations of words, consideration of edge embeddings, and development of estimation methods for complex word hierarchies. The results confirm the prospects of the approach for creating adaptable models capable of processing multi-format documents (scientific articles, legal texts). This paper highlights the need for further research in the field of regularization and extension of training data, opening up ways to improve the portability of layout analysis methods to new domains. The code and models were published on GitHub (https://github.com/YRL-AIDA/wordGLAM).

Keywords: graph neural networks; convolutional graph neural networks; document segmentation; document layout analysis; document regions; document blocks; document segments.

For citation: Kopylov D.E., Mikhailov A.A., Trifonov R.I. Segmentation of documents based on graph neural networks: from strings to words. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part1, 2025, pp. 219-232 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-14.

Acknowledgements. The research was carried out within the state assignment of Ministry of Science and Higher Education of the Russian Federation (theme No. 1023110300006-9).

1. Введение

Современные предприятия и организации ежедневно сталкиваются с огромными объемами документов, большая часть которых хранится в формате PDF. Этот формат, разработанный для кроссплатформенного представления данных, существует в двух основных вариантах: 1) в виде изображения (растрового/векторного) без семантической разметки; 2) в виде структурированного файла, содержащего текст, метаданные, векторную графику и инструкции для визуализации, что обеспечивает точное воспроизведение макета на любых устройствах. Несмотря на свою универсальность, PDF документы остаются сложными для автоматизированного извлечения структурированной информации, что порождает необходимость применения методов анализа макета документа (Document Layout Analysis, DLA). Хорошими обзорами указанной проблемы являются [1] и [2].

Анализ макета документа направлен на автоматическое обнаружение и классификацию семантических элементов, таких как заголовки, абзацы, таблицы, изображения и списки. Как

правило, он предшествует решению других задач, например, поиска информации по документам, пересказа или вопросно-ответных систем для документов.

Традиционные подходы к анализу макета документов можно разделить на методы, основанные на правилах (rule-based), и системы использующие нейронные сети. Пионерские работы, например, работа [3], применяли правила, учитывающие расстояние между блоками, выравнивание и характеристики шрифтов, что требует ручной настройки правил для каждого типа документов и ограничивает адаптивность к разнообразным макетам. С другой стороны, нейронные сети, особенно основанные на архитектуре Трансформер (см., например, [4-7]), обрабатывают документ как изображение, набор текстовых блоков или комбинацию изображения и текста. На текущий момент они демонстрируют высочайший уровень качества в решении задач, связанных с анализом макетов документов.

Насколько известно авторам, лучшим результатом на наборе данных [8] остается модель Vision Grid Transformer (VGT) из работы [6], ее оценка по метрике mAP@IoU[0.5:0.95] составляет 0.962. Однако ключевым недостатком модели является ее плохая переносимость на новые домены данных.

В этом контексте перспективными представляются модели, основанные на графовых нейронных сетях (Graph Neural Networks, GNN) [9-14]. Основное преимущество GNN заключается в их способности моделировать топологические зависимости между элементами, что соответствует природе документов. Использование графа для представления документа не является современным подходом, а отсылает к идеям, ранее высказанным в работах [15-17].

Ключевой проблемой многих современных подходов к сегментации документов с использованием GNN является использование строк [10-11] или локальных областей [9] как узлов графа. Строки в сложных документах, таких как научные статьи или юридические тексты, часто детектируются некорректно из-за разнообразия форматов, шрифтов и структуры. Фрагментарность небольших областей затрудняет установление семантических связей между ними. В отличие от этих подходов, использование слов в качестве узлов графа позволяет достичь баланса между детекцией и связностью. Слова детектируются легче, чем строки, так как они имеют более четкие границы и меньше зависят от форматирования. Кроме того, их можно агрегировать в строки, абзацы или разделы, сохраняя иерархию документа. Таким образом, слова являются «золотой серединой», сочетающей точность детекции с возможностью моделирования структурных зависимостей, что делает их более подходящими для задач сегментации документов.

2. Исследуемая архитектура модели

В данном разделе представлена предлагаемая модель сегментации документов на основе GNN. Первый подраздел описывает структуру входных данных, где узлами графа являются слова. Второй подраздел детализирует архитектуру сети, включая модификации базового подхода из работы [12].

2.1 Графы

Все GNN работают непосредственно с графами, представленными в виде матриц. Граф, как известно, характеризуется парой (V, E), где V – множество узлов, E – множество ребер. В качестве узлов выступают вещественные векторы $v_i \in R^m, i = 1,2,\dots,n$. Эти векторы могут быть получены из участков документа, в частности, регионов, строк, слов. Ребра между узлами, часто кодируются в виде матрицы смежности $A \in \{0,1\}^{n\times n}$, где единицы характеризуют наличие связи, а номера строк и столбцов – номера соответствующих узлов. В качестве альтернативы используется Лапласиан $L = D - A \in Z^{n\times n}$ (целочисленная матрица), где $D = diag\{d_i\}$ – диагональная матрица, где d_i – степень i–го узла. Обе матрицы, как правило, разреженные и хранятся в компактном виде (в данной работе матрица

смежности хранится как массив пар из двух индексов). В добавок к матрице формируется множество векторов $e_i \in R^l, i = 1,2,\dots,k$, содержащим признаки для каждого ребра.

Процесс построения графа начинается с детекции слов. Отметим, что документ может не иметь текстовый слой. Если документ представлен в виде изображения, для детекции слов применяется OCR Tesseract [18], который распознаёт текст и определяет координаты ограничивающих рамок (bounding boxes, bbox) для каждого слова. Для PDF документов с текстовым слоем, текст и сопутствующие метаданные извлекаются при помощи PDF парсера [19]. Вопрос распознавания слов является отдельной задачей и выходит за рамки настоящей работы.

Ключевое отличие от работы [12] заключается в использовании слов вместо строк в качестве узлов. Это обусловлено тем, что строки в документах (например, таблицах или диаграммах) часто содержат ошибки детекции или не несут самостоятельной семантики.

Каждое слово представляется в виде узла графа с тремя группами признаков:

1. Геометрические: координаты левого верхнего угла bbox'а, координаты правого нижнего угла bbox'а, высота и ширина;
2. Текстовые: первые 32 координаты из векторного представления слова (используется базовая многоязычная модель BERT с учетом регистра [4], который возвращает векторы длиной 512), индикаторы ключевых слов (для пяти классов), индикатор маркированного и нумерованного списка, индикатор знаков препинаний («.», «;», «,», «:»);
3. Стилевые: векторное представление шрифта размерности 3, полученное с использованием сверточной модели, обученной на генерированных данных распространенных шрифтов.

Для построения ребер применяется модифицированный метод k-ближайших соседей (k=4). Для каждого узла выбираются четыре соседа: ближайшие слева, справа, сверху и снизу [20]. Расстояние вычисляется от границ ограничивающих рамок слов с учетом направления поиска. Увеличение значения k усугубляет проблему, описанную в разделе 5.1. Альтернативные методы, такие как триангуляция Делоне, не улучшают результаты, но значительно увеличивают вычислительную сложность.

Данный подход интегрирует локальные атрибуты слов с глобальной структурой документа, учитывая как пространственные, так и семантические связи.

2.2 Архитектура графовой нейронной сети

Представленная в настоящей работе модель обозначается как WordGLAM. Наименование данной модели отсылает к модели GLAM, описанной в источнике [12], архитектура которой была взята за основу. Архитектура WordGLAM представлена на рис. 1.

Для реализации модели использовалась библиотека PyTorch 2.6.0 и расширение PyG 2.6.1.

Основу архитектуры составляют слои TagConv (Topology Adaptive Graph Convolutional Networks) [21], являющиеся обобщением идеи обычного сверточного слоя в GNN [22]. Слои TagConv чередуются с линейными слоями. Слои TagConv выполняют свертку на графе, учитывая топологию и признаки соседних узлов, что позволяет эффективно агрегировать информацию о локальной структуре графа. Математически слой TagConv можно записать в следующем виде:

$$X_{j+1} = \sum_{k=0}^{K} \left(D^{\frac{-1}{2}} A D^{\frac{-1}{2}} \right)^k X_j W_{jk},$$

где X_j – матрица векторных представлений узлов на j–м слое, D – диагональная матрица

степеней узлов (числом связей у каждого элемента), W_{jk} – обучаемые веса. Параметр K – характеризует глубину агрегации признаков (для всех слоев выбран $K = 2$). Число слоев TagConv и значение параметра K подбирались экспериментально (подробнее в подразделе 4.3).

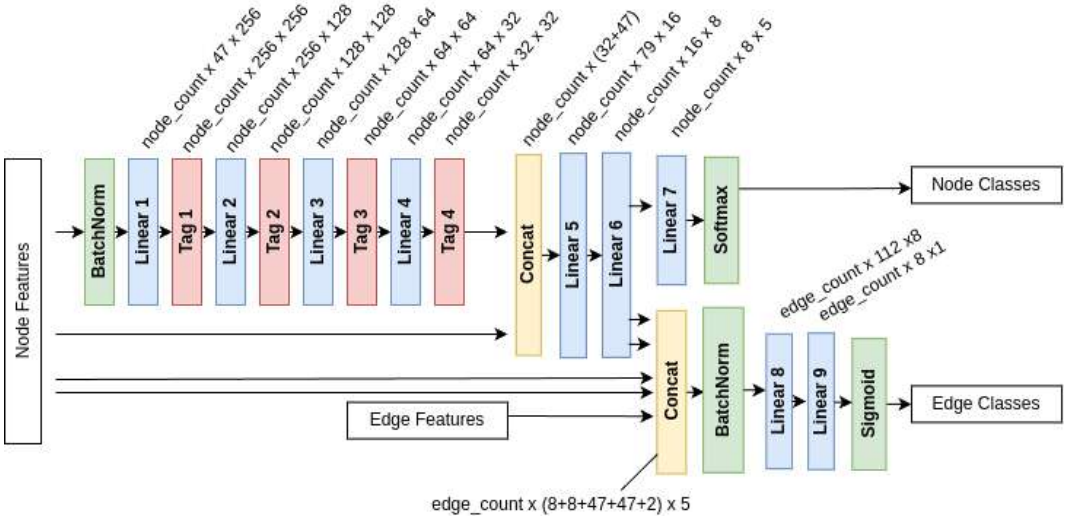


Рис. 1. Архитектура WordGLAM.
Fig. 1. WordGLAM architecture.

На вход первой части модели подаются признаки узлов, сгруппированные в порции данных. Каждая порция подвергается процедуре компонентной нормализации. Данный процесс включает в себя расчет разницы между исходным значением и ее математическим ожиданием, а также последующее деление полученной разницы на ее среднеквадратичное отклонение. Такая нормализация позволяет стабилизировать процесс обучения и улучшить обобщающую способность модели. Выходом первой части являются обогащенные представления узлов, учитывающие их локальный контекст, которые затем передаются на следующий этап обработки.

Вторая часть оценивает значимость ребер. Для каждого ребра входной вектор формируется конкатенацией: исходных признаков связанных узлов, их обогащенных представлений и признаков ребер (длины и наклона). Вторая часть модели на входе также работает с группированными в порцию векторами, которые предварительно проходят компонентную нормализацию. После нормализации векторы проходят через каскад полносвязных слоев, так чтобы на выходе каждый вектор, характеризующий ребро, преобразовывался скалярное значение. Это значение определяет вероятность того, что ребро соединяет два слова внутри региона, а не слова из разных регионов.

Между слоями в обеих частях модели используется нелинейная активация GeLU [23].

Применение такой архитектуры позволяет эффективно выделять компоненты связности в графе, которые соответствуют отдельным блокам текста. Удаление ребер, классифицированных как лишние, приводит к разбиению исходного графа на подграфы, каждый из которых представляет собой отдельный текстовый блок. Это обеспечивает не только сегментацию текста, но и возможность его дальнейшей классификации на основе структуры графа и признаков узлов. Для каждого подграфа строится ограничивающая рамка, которая и является блоком. Если блоки пересекаются, то они объединяются в один (данный вопрос обсуждается в подразделе 5.1).

3. Данные и метрика качества

В работе используется набор данных PubLayNet [8], ставший основным для задачи восстановления макета документа. Для подбора гиперпараметров при обучении использовались 1000 документов из обучающей выборки PubLayNet (из них 10% использовались для валидации модели), при тестировании использовались 50 документов из валидационной выборки PubLayNet. Финальное обучение проводилось на всем наборе данных (также 10% использовалось для валидации модели), а тестирование выполнялось на 200 документах из валидационной выборки.

Для оценки моделей детекции традиционно применяют метрику mAP@IoU [0.5:0.95]. Качество детекции одного региона оценивается с помощью IoU (Intersection over Union) – отношения площади пересечения верного и предсказанного региона к площади их объединения. Для подсчета общей точности используется mAP (Mean Average Precision) – площадь под кривой на графике с полнотой (Recall) в качестве абсциссы и точностью (Precision) в качестве ординаты. Эта кривая строится путем варьирования порога IoU (от 0.5 до 0.95 с шагом 0.05). Более подробно с метрикой можно ознакомиться, например, в работе [24].

Критерий IoU основан на площади пересечения и объединения. Для регионов текста такой критерий не отражает семантической целостности региона. К ошибкам в семантике можно отнести ложные штрафы за неточные границы, даже если все слова региона попали в него. Другой пример, когда при разделении одной и той же строки в регионах с одним и тем же текстом, но разными межстрочными интервалами оценки будут разными, хотя семантически ошибка одна и та же.

Общий результат подсчитывается с использованием метрики mAP с заданным порогом. В своем расчете метрика учитывает уверенность модели в наличие региона, который она детектировала. На практике уверенность модели заменяется фиксированным порогом, что делает оценки метрик не показательными для практических задач.

Авторами в качестве критерия предлагается рассматривать не отношение площадей, а отношение числа слов WordIoU. В качестве самой оценки, используется классическая F1-мера с двумя порогами: 0.5 и 0.95 (оптимистичный и пессимистичный).

$$WordIoU = \frac{count(W_{true} \cap W_{pred})}{count(W_{true} \cup W_{pred})},$$

где W_{true}, W_{pred} – слова находящиеся в правильно размеченном и предсказанном регионах соответственно. Далее в таблицах указана и классическая метрика mAP@IoU [0.5:0.95], которая по мнению авторов не показывает объективно качество для документов.

4. Обучение модели

В данном разделе, перед тем как перейти непосредственно к обучению модели на всем наборе данных (подраздел 4.4), обсуждается важный для нашей модели вопрос балансировки данных (подраздел 4.1), функции потерь (подраздел 4.2), подбор числа слоев и глубины агрегации (подраздел 4.3). Для подбора гиперпараметров везде использовался оптимизатор Adam с темпом обучения (learning rate), равным 5×10^{-3}. Обучение проходило в течение 30 эпох, веса обновляются через каждые 80 документов.

4.1 Балансировка данных

При сегментации на уровне строк распределение связей внутри и между регионами близко к сбалансированному. В случае выстраивания графа из слов, это становится не так. Однако при использовании слов в качестве узлов доля внутрирегиональных связей снижается в 5 раз по сравнению с межрегиоными, что создает дисбаланс классов. Для компенсации дисбаланса

применен метод взвешивания классов (class weighting). Для положительного класса, когда связь внутри региона (оба слова из одного региона), ставится вес равный 0.15. Аналогичный дисбаланс обнаружен в задаче классификации узлов, где также использовано взвешивание (для изображений вес равен 2.63, для текста 0.015, для заголовка 0.946, для списка 1.268, для таблицы 0.136). Результаты обучения сравнения моделей с балансировкой и без нее представлены в табл. 1.

Табл. 1. Важность балансировки данных для обучения модели.
Table 1. The importance of balancing data for model training.

Балансировка	**F1@WordIoU[0.5]**	**F1@WordIoU[0.95]**	**mAP@IoU[0.5:0.95]**
Нет	0.1103	0.0441	0.0050
Есть	0.5257	0.3457	0.0834

4.2 Функция потерь

Обучение исходной модели GLAM и новой модели WordGLAM осуществлялось с использованием комбинированной функции потерь, которая совмещает задачу классификации ребер и узлов. Функция потерь определяется как:

$$Loss = \alpha Loss_{node} + (1 - \alpha) Loss_{edge},$$

где

- $Loss_{edge}$ – бинарная кросс-энтропия для классификации ребер;
- $Loss_{node}$ – кросс-энтропия для классификации узлов;
- α – весовой коэффициент.

В модели GLAM совмещаются задачи классификации и сегментации регионов. На данном этапе решается только задача сегментации, одна из причин описана в подразделе 5.4. Тем не менее, функция потерь представляет собой композицию функций $Loss_{edge}$ и $Loss_{node}$. При сегментации используются векторные представление узлов, и тем самым классификация выступает некоторой регуляризацией. За счет такого штрафа модель строит векторное представление узлов не только для решения задачи сегментации, а в целом возвращает векторы, которые содержат информацию о словах, в частности, к какому классу (к тексту, списку, заголовку или таблице) они относятся.

Был проведен эксперимент для разных значений α, который показал необходимость наличия данного слагаемого (табл. 2).

Табл. 2. Эксперимент с параметром функции потерь.
Table 2. Experiment with the parameter of the loss function.

α	**F1@WordIoU[0.5]**	**F1@WordIoU[0.95]**	**mAP@IoU[0.5:0.95]**
0.05	0.1536	0.0607	0.0060
0.25	0.1780	0.0803	0.0097
0.50	0.4083	0.2485	0.0526
0.75	**0.5257**	**0.3457**	**0.0835**
0.85	0.4950	0.3137	0.0630
0.95	0.0053	0.0011	0.0000

4.3 Подбор числа слоев и глубины агрегации

В архитектуре WordGLAM ключевую роль играют слои TagConv, эффективность которых критически зависит от глубины агрегации соседей (K) и количества слоев. Оптимальные значения этих параметров зависят от сложной иерархической структуры обрабатываемых документов. Слишком малое K или недостаточное число слоев ограничивает охват контекста,

лишая модель преимуществ графового подхода. Чрезмерно большое K или избыток слоев может привести к «размытию» полезной информации в графе и переобучению. По результатам анализа данных (табл. 3) было принято решение использовать 4 слоя с параметром K=2.

4.4 Обучение модели на всем наборе данных

В отличие от экспериментов для подбора гиперпараметров, которые осуществлялись на 1000 документах, а веса модели обновлялись каждые 80 документов, в эксперименте по обучению модели на всем наборе данных веса обновлялись после каждых 500 документов. Все остальные параметры в эксперименте остаются неизменными. Поведение функции потерь при обучении представлено на рис. 2. Несколько примеров обработанных документов приведены на рис. 3. Сравнения с моделью VGT [6] (модель возвращает уровень доверия, поэтому берутся только те регионы, которые имеют уровень доверия больше 0.5, и считаем, что они имеют уровень доверия равный 1), приводится в табл. 4. Модель WordGLAM значительно уступает лидирующей модели VGT по точности. При увеличении объема обучающей выборки наблюдается значительное снижение точности по всем метрическим показателям, что превышает двукратное уменьшение. Причины такого поведения требуют дальнейшего исследования.

Табл. 3. Значения метрик mAP@IoU[0.5:0.95] и F1@WordIoU[0.5] в зависимости от числа слоев (строки) и глубины K (столбцы).
Table 3. mAP@IoU[0.5:0.95] and F1@WordIoU[0.5] Metrics vs. Number of Layers (Rows) and Depth (Columns).

		Глубина (K)			
		2	**3**	**4**	**5**
Число слоев	**1**	0.0543/0.4595	0.0546/0.4422	0.0379/0.4211	0.0449/0.3775
	2	0.0504/0.4162	0.0478/0.4332	0.0601/0.4609	0.0478/0.4598
	3	0.0475/0.4401	0.0503/0.4454	0.0488/0.4231	0.0486/0.3662
	4	**0.0770/0.5262**	0.0486/0.4249	0.0465/0.4090	0.0551/0.4464
	5	0.0468/0.4036	0.0314/0.3609	0.0568/0.4370	0.0623/0.4849

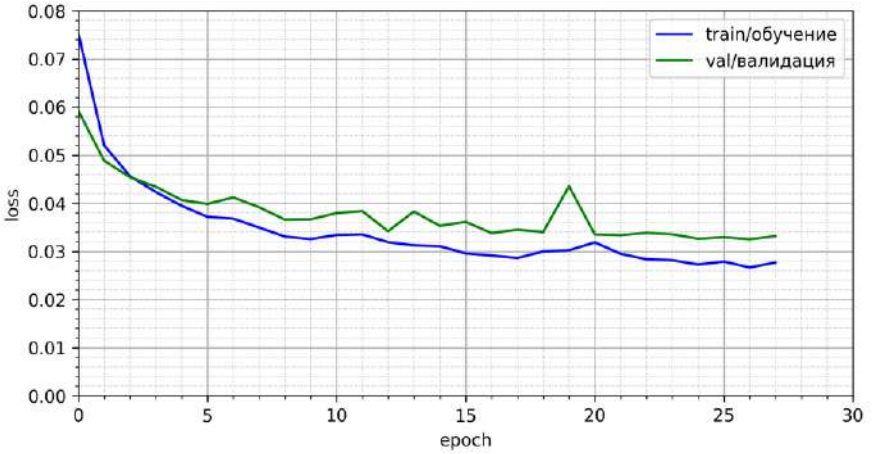


Рис. 2. Обучение WordGLAM на всем наборе данных.
Fig. 2. Learning Word GLAM on the entire dataset.

5. Трудности при работе с документом, представленным в виде графа слов

При переходе с уровня строк на уровень слов возник ряд трудностей, не все из которых удалось решить. Для части из них удалось только сформулировать проблемы. В этом разделе перечислены открытые вопросы, которые возникают из-за того, что происходит переход на более низкий уровень – уровень слов.

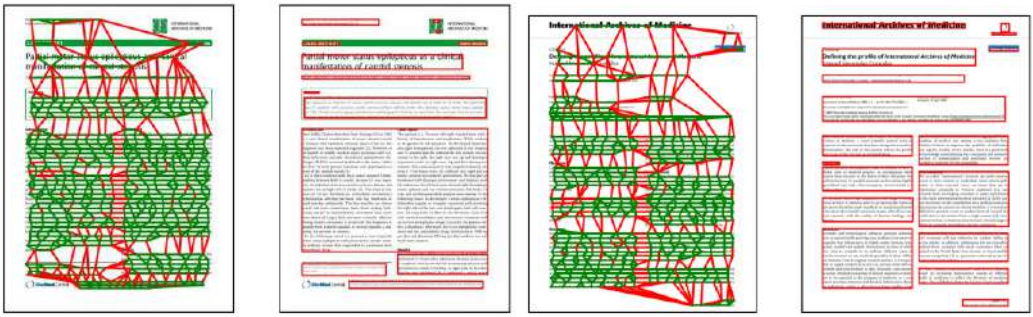


Рис. 3. Пример работы WordGLAM.
Fig. 3. An example of how WordGLAM works.

Табл. 4. Сравнение WordGLAM с VGT [6].
Table 4. Comparing WordGLAM with VGT [6].

	mAP@IoU [0.5:0.95]	**F1@WordIoU [0.95]**	**F1@WordIoU [0.5]**
VGT (весь PubLayNet)	0.8713	0.8613	0.9985
WordGLAM (1000 документов из PubLayNet)	0.0835	0.3457	0.5257
WordGLAM (весь PubLayNet)	0.0182	0.1520	0.2645

5.1 «Мосты» и «цепная связь» между регионами

Несмотря на высокую точность обнаружения связей (рис. 4), алгоритм демонстрирует значительные ошибки в сегментации регионов. Основные проблемы связаны с двумя явлениями: «мостами» и «цепными связями».

«Мост» возникает, когда алгоритм ошибочно идентифицирует связь между двумя независимыми регионами, объединяя их в один компонент. Например, на рис. 4 два нижних региона (заголовок и текст) сливаются в единый регион из-за единственной ложной связи. «Цепная связь» – каскадное распространение ошибки объединения на соседние регионы. На том же примере слияние нижних блоков запускает цепную реакцию: алгоритм последовательно объединяет все пересекающиеся регионы, начиная с неверно найденного региона, отмеченного фиолетовой рамкой, который объединяется с регионом в синей рамке в один общий регион. В результате 5 корректно выделенных изначально регионов из-за «цепной связи» объединились в один.

Попытка исправить проблему через запрет объединения регионов потребует введения множества эвристик (например, правил для конкретных типов пересечений), что снизит универсальность обработчика. Альтернативой может стать регуляризация, учитывающая: положение центров блоков (минимизация расстояний между центрами связей) и геометрическую форму (штраф за отклонение от «прямоугольности»). Однако авторам не удалось найти способ для такой регуляризации, что открывает направление для новых исследований.

5.2 Текст или абзацы, таблицы или колонки

Основу геометрического графа составляют пространственные признаки элементов: их координаты, визуальная схожесть (шрифт) и расстояние между ними. Эти параметры позволяют группировать элементы в регионы. Однако ключевая сложность заключается в том, что пространственные параметры границы (например, интервалов между абзацами) нарушает корректность сегментации.

Главная проблема при работе с текстом – ложное объединение абзацев. Если между ними нет выраженных интервалов, геометрический граф интерпретирует их как единый блок, несмотря на смысловую разрозненность (рис. 5). Например, два независимых абзаца, оформленных одинаково, будут слиты в один регион. Такие ошибки мало влияют на понимание содержания, но критичны для метрик качества сегментации (например, IoU), которые не учитывают семантику. Это демонстрирует необходимость разработки метрик, оценивающих не только геометрию, но и контекст.

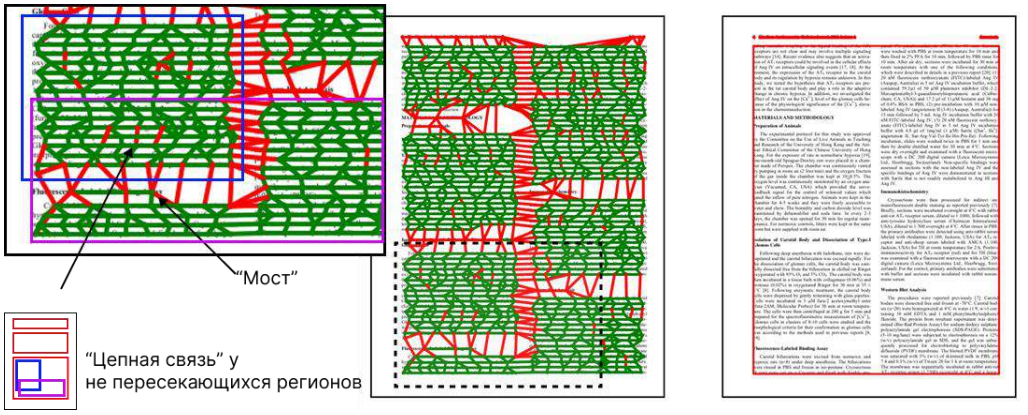


Рис. 4. «Мосты» и «цепные связи».
Fig. 4. «Bridges» and «Chain links».

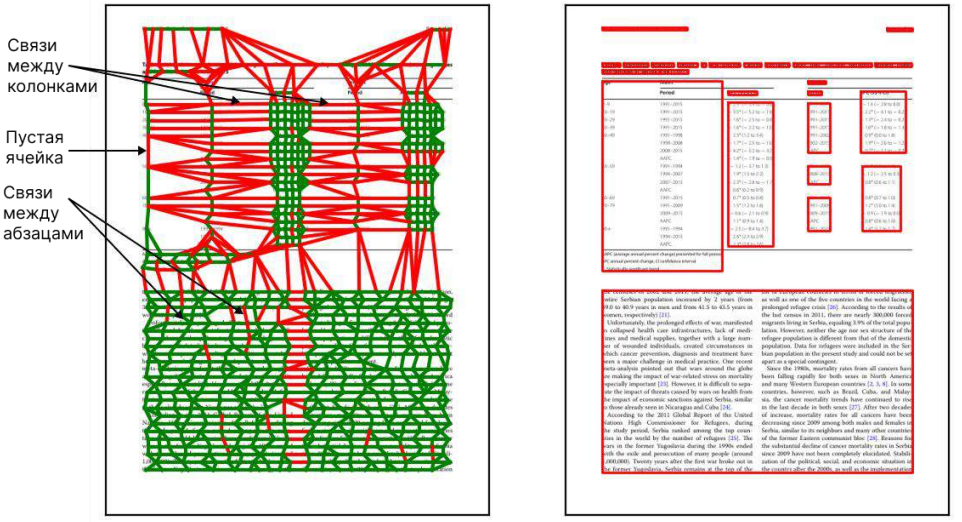


Рис. 5. Связи между колонками и абзацами.
Fig. 5. Edges between columns and paragraphs.

Для таблиц характерна обратная проблема: из-за разнородных отступов, пустых ячеек и отсутствия визуальных разделителей выделяют изолированные блоки вместо единой структуры (рис. 5). Например, пустая ячейка может трактоваться как граница между столбцами, что разбивает таблицу на несвязанные фрагменты. Это усложняет обработку документов со сложной сеточной структурой, где геометрические признаки не всегда отражают логические связи.

Решение проблемы возможно через расширение спектра классифицируемых объектов. В исходной обучающей выборке было представлено пять типов меток. Для достижения этой цели можно использовать наборы данных с большим количеством типов меток, такие как DocLayNet [25] (11 типов) или M6Doc [26] (74 типа).

5.3 Классификация узлов без признаков ребер

Если документ разбивается на строки, то узлы сами по себе плохо характеризуют блок, чего нельзя сказать о ребрах между словами [20]. Даже визуально таблицы имеют вид прямоугольной сетки; текст имеет горизонтальные линии строк и случайные связи между строками; список будет иметь горизонтальную линию объединяя цифры и маркеры, а остальное как у текста; заголовок в большинстве случаев состоит из одной линии.

В данной архитектуре при классификации не используется информация о ребрах. Возможным улучшением будет являться передача признаков для удаления ребер для классификации.

Другой вариант – обучение модели, которая после сегментации классифицирует граф региона целиком. Такой подход, с одной стороны, разбивает задачу на две, но, с другой стороны, обучение происходит не в общем контексте документов.

5.4 Векторное представление узлов

В модели TAGConv ключевые атрибуты передаются в виде векторного представления узлов. На каждом уровне осуществляется процесс агрегации. В случае признаков стиля, координат и индикаторов их семантика понятна. Однако возникает вопрос о сущности агрегированного вектора слова. Если граф отражает семантические связи между словами, то такой вектор имеет обоснованное значение. Однако в данной модели связи формируются на основе позиции элементов в структуре, и слова не обладают логической взаимосвязью (за исключением случаев, когда они являются элементами таблицы или текстовой строки).

Текстовые признаки формируются с использованием модели и BERT [4]. Для оптимизации размера признакового вектора вместо полных 512 компонентов используются только первые 32, что не превышает удвоенный размер остальных признаков. Эксперимент с полным вектором (512 компонентов) показали значение метрики mAP@IoU[0.5:0.95] = 0.0223. Вопрос об оптимальном способе интеграции текстовой информации требует дальнейшего изучения.

6. Заключение

В данной работе предложен альтернативный подход к сегментации документов на основе графовых нейронных сетей (GNN), где в качестве узлов графа используются слова. Несмотря на то, что текущие результаты модели уступают по метрикам современным подходам, опирающимся на строки или локальные области, метод демонстрирует значительный исследовательский потенциал. Использование слов как базовых элементов позволило выявить ранее незаметные проблемы, связанные с балансировкой данных, регуляризацией функции потерь и оценкой качества моделей в области восстановления логической структуры документов.

Архитектура модели, основанная на модификациях графовых сверточных слоев из работы

[12], хотя и не превзошла существующие аналоги, подтвердила возможность построения иерархических структур документа через объединение слов. Модель требует дальнейшей настройки гиперпараметров и расширения обучающей выборки для улучшения обобщающей способности.

Еще одним вкладом работы стало обнаружение новых исследовательских вызовов, связанных с переходом на уровень слов: например, проблема «мостов» и «цепных связей» при кластеризации; вопрос правильного выбора классов при разметке данных; вопрос о том, что более точно характеризует регион: слова как множество или их взаимное расположение. Эти проблемы, ранее не акцентированные в литературе, открывают направления для будущих исследований.

Несмотря на полученные результаты, предложенная модель WordGLAM закладывает основу для более гибкого анализа документов с возможностью переносимости модели на новые домены данных. Ее развитие, включая улучшение векторного представления слов и учета векторного представления ребер при классификации, может привести к прорыву в задачах обработки документов.

Список литературы / References

[1]. Kise K. Page Segmentation Techniques in Document Analysis. In: Doermann, D., Tombre, K. (eds) Handbook of Document Image Processing and Recognition, 2014, Springer, London, pp. 135-175. DOI: 10.1007/978-0-85729-859-1_5.

[2]. BinMakhashen G. M., Mahmoud S. A. Document Layout Analysis: A Comprehensive Survey. ACM Computing Surveys (CSUR), vol. 52, issue 6, pp. 1-36. DOI:10.1145/3355610.

[3]. Tsujimoto S., Asada H. Major components of a complete text reading system. In Proc. of the IEEE, 1992, 80(7), pp. 1133-1149. DOI: 10.1109/5.156475.

[4]. Koroteev M. V. BERT: a review of applications in natural language processing and understanding. CoRR, vol. abs/2103.11943, 2021 [Online]. Available at: https://arxiv.org/abs/1810.04805.

[5]. Huang Y., Lv T., Cui L., Lu Y., Wei F. LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking. Proc. of the 30th ACM International Conference on Multimedia, 2022, pp. 4083-4091.DOI: 10.1145/3503161.3548112.

[6]. Da C., Luo C., Zheng Q., Yao C. Vision Grid Transformer for Document Layout Analysis. IEEE/CVF International Conference on Computer Vision (ICCV), Paris, France, 2023, pp. 19405-19415, DOI: 10.1109/ICCV51070.2023.01783.

[7]. Sun T., Cui C., Du Y., Liu Y. PP-DocLayout: A Unified Document Layout Detection Model to Accelerate Large-Scale Data Construction. CoRR, vol. abs/2503.17213, 2025 [Online]. Available at: https://arxiv.org/abs/2503.17213.

[8]. Zhong X., Tang J., Jimeno-Yepes A. PubLayNet: Largest Dataset Ever for Document Layout Analysis. 2019 International Conference on Document Analysis and Recognition (ICDAR), 2019, pp. 1015-1022. DOI: 10.1109/ICDAR.2019.00166.

[9]. Maia A. L. L. M., Julca-Aguilar F. D. Hirata N. S. T. A Machine Learning Approach for Graph-Based Page Segmentation. In Proc. 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI), Parana, Brazil, 2018, pp. 424-431. DOI: 10.1109/SIBGRAPI.2018.00061.

[10]. Wang R., Fujii Y., Popat A.C. General-Purpose OCR Paragraph Identification by Graph Convolution Networks. CoRR, vol. abs/2101.12741, 2021 [Online]. Available at: https://arxiv.org/abs/2101.12741.

[11]. Wei S., Xu N. PARAGRAPH2GRAPH: A GNN-based framework for layout paragraph analysis. CoRR, vol. abs/2304.11810, 2023 [Online]. Available at: https://arxiv.org/abs/2304.11810.

[12]. Wang, J. et al. (2023). A Graphical Approach to Document Layout Analysis. Proc. of the 17th ICDAR, 2023, vol. 14191, pp. 53-69. DOI:10.1007/978-3-031-41734-4_4.

[13]. Dai HS., Li XH., Yin, F., Yan, X., Mei, S., Liu, CL. (2024). GraphMLLM: A Graph-Based Multi-level Layout Language-Independent Model for Document Understanding. Proc. of the 18th ICDAR, 2024, vol 14804, pp. 227-243. DOI: 10.1007/978-3-031-70533-5_14.

[14]. Chen Y. et al. Graph-based Document Structure Analysis. CoRR, vol. abs/2502.02501, 2025 [Online]. Available at: https://arxiv.org/abs/2502.02501.

[15]. O'Gorman L. The document spectrum for page layout analysis. IEEE Transactions on Pattern Analysis and Machine Intelligence, 15 (11), pp. 1162-1173, 1993. DOI: 10.1109/34.244677.

[16]. Kise K., Sato A. Iwata M. Segmentation of Page Images Using the Area Voronoi Diagram. Comput. Vis. Image Underst, 1998, vol. 70, pp. 370-382. DOI:10.1006/cviu.1998.0684.
[17]. Yi Xiao and Hong Yan. Text region extraction in a document image based on the Delaunay. Pattern Recognit, 2003, vol. 36, pp. 799-809. DOI: 10.1016/S0031-3203(02)00082-1.
[18]. Tesseract User Manual, Available at: https://tesseract-ocr.github.io/tessdoc, accessed 5.08.2025.
[19]. PrecisionPDF, Available at: https://github.com/YRL-AIDA/PrecisionPDF, accessed 5.08.2025.
[20]. Kopylov D., Mikhaylov A. How To Classify Document Segments Using Graph Based Representation and Neural Networks. Ivannikov Memorial Workshop (IVMEM), 2024, pp. 36-41. DOI: 10.1109/IVMEM63006.2024.10659393.
[21]. Du J., Zhang S., Wu G. Moura J. M. F., Kar S. Topology Adaptive Graph Convolutional Networks. CoRR, vol. abs/1710.10370, 2018 [Online]. Available at: https://arxiv.org/abs/1710.10370.
[22]. Kipf T. N., Welling M. Semi-Supervised Classification with Graph Convolutional Networks. CoRR, vol. abs/1609.02907, 2017 [Online]. Available at: https://arxiv.org/abs/1609.02907.
[23]. Hendrycks D., Gimpel K. Gaussian error linear units (GELUs). CoRR, vol. abs/1606.08415, 2016 [Online]. Available at: https://arxiv.org/abs/1606.08415.
[24]. Everingham M., Van Gool L., Williams C.K.I. et al. The Pascal Visual Object Classes (VOC) Challenge. International Journal of Computer Vision, 2010, vol. 88, pp. 303–338. DOI: 10.1007/s11263-009-0275-4.
[25]. Pfitzmann B., Auer C., Dolfi M., Nassar A. S., Staar P. DocLayNet: A Large Human-Annotated Dataset for Document-Layout Analysis. In Proc. of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22), 2022, pp. 3743-3751. DOI: 10.1145/3534678.3539043.
[26]. Cheng H. et al. M6Doc: A Large-Scale Multi-Format, Multi-Type, Multi-Layout, Multi-Language, Multi-Annotation Category Dataset for Modern Document Layout Analysis. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2023, pp. 15138-15147. DOI: 10.1109/CVPR52729.2023.01453.

Информация об авторах / Information about authors

Даниил Евгеньевич КОПЫЛОВ – магистрант направления подготовки «Прикладная математика и информатика» Иркутского государственного университета, сотрудник Института динамики систем и теории управления имени В.М. Матросова Сибирского отделения Российской академии наук. Сфера научных интересов: прикладная математика, анализ данных.

Daniil Evgenievich KOPYLOV is a master student of Irkutsk State University, employee of Matrosov Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences. Research interests: applied mathematics, data analysis.

Андрей Анатольевич МИХАЙЛОВ является заведующим Молодежной лаборатории Искусственного интеллекта, обработки и анализа данных Института динамики систем и теории управления имени В.М. Матросова. Сфера научных интересов: анализ электронных документов, распознавание образов.

Andrey Anatolievitch MIKHAYLOV is the head of the Youth laboratory of AI, Data Processing and Analysis of Matrosov Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences. His research interests include document analysis, image recognition.

Роман Игоревич ТРИФОНОВ – студент направления подготовки «Фундаментальная информатика и информационные технологии» Иркутского государственного университета, сотрудник Института динамики систем и теории управления имени В.М. Матросова Сибирского отделения Российской академии наук. Сфера научных интересов: прикладная информатика, анализ данных, нейронные сети.

Roman Igorevich TRIFONOV is a student of Irkutsk State University, employee of Matrosov Institute for Systems Dynamics of and Control Theory of Siberian Branch of Russian Academy of Sciences. Research interests: applied informatics, data analysis, neural networks.

DOI: 10.15514/ISPRAS-2025-37(6)-15

Сравнение интерпретируемости моделей ResNet50 и ViT-224 в задаче классификации бактерий на снимках сканирующего электронного микроскопа

В.Н. Гридин, ORCID: 0000-0002-6361-9113 <info@ditc.ras.ru>
И.А. Новиков, ORCID: 0000-0003-4898-4662 <i.novikov@niigb.ru>
Б.Р. Салем, ORCID: 0000-0003-3348-9407 <isub97@gmail.com>
В.И. Солодовников, ORCID: 0000-0001-5533-214X <v_solodovnikov@hotmail.com>
Центр информационных технологий в проектировании РАН,
Россия, 143003, Московская обл., г. Одинцово, ул. Маршала Бирюзова, д. 7А.

Аннотация. В работе проведено исследование интерпретируемости двух популярных архитектур глубокого обучения ResNet50 и Vision Transformer (ViT-224) в рамках решения задачи классификации патогенных микроорганизмов на изображениях, полученных посредством сканирующего электронного микроскопа и предварительной пробоподготовкой с использованием лантаноидного контрастирования. Помимо стандартных показателей качества, таких как: точность, полнота и F1-мера, ключевым аспектом стало исследование встроенных карт внимания Vision Transformer и пост-интерпретации работы обученной модели ResNet50 с помощью метода Grad-CAM. Эксперименты выполнялись на исходном наборе данных, а также трёх его модификациях: с обнулённым фоном (threshold), с модифицированными участками изображения методом inpainting, и с полностью очищенным фоном с помощью обнуления фоновых участков. Для оценки универсальности механизма внимания в Vision Transformer дополнительно проведён тест на классической задаче распознавания рукописных цифр MNIST. Результаты показали, что архитектура Vision Transformer демонстрирует более локализованные и биологически обоснованные тепловые карты внимания, а также большую устойчивость к изменению фонового шума.

Ключевые слова: архитектура Vision Transformer; модель ResNet50; метод Grad-CAM; карты внимания; тепловые карты внимания; интерпретируемость; классификация; бактерии; анализ изображений.

Для цитирования: Гридин В.Н., Новиков И.А., Салем Б.Р., Солодовников В.И. Сравнение интерпретируемости моделей ResNet50 и ViT-224 в задаче классификации бактерий на снимках сканирующего электронного микроскопа. Труды ИСП РАН, том 37, вып. 6, чать 1, 2025 г., стр. 233-242. DOI: 10.15514/ISPRAS–2025–37(6)–15.

Благодарности: Исследование выполнено в рамках темы № FFNR-2024-0003.

Comparison of the Interpretability of ResNet50 and ViT-224 Models in the lassification Task is Erroneous on Images of a Scanned Microscope Object

V.N. Gridin, ORCID: 0000-0002-6361-9113 <info@ditc.ras.ru>
I.A. Novikov, ORCID: 0000-0003-4898-4662 <i.novikov@niigb.ru>
B.R. Salem, ORCID: 0000-0003-3348-9407 <isub97@gmail.com>
V.I. Solodovnikov, ORCID: 0000-0001-5533-214X <v_solodovnikov@hotmail.com>
Design Information Technologies Center Russian Academy of Sciences,
7A, Marshal Biryuzova str., Odintsovo, Moscow Oblast, 143003, Russia.

Abstract. The paper studies the interpretability of two popular deep learning architectures, ResNet50 and Vision Transformer (ViT-224), in the context of solving the problem of classifying pathogenic microorganisms in images obtained using a scanning electron microscope and preliminary sample preparation using lanthanide contrast. In addition to standard quality metrics such as precision, recall, and F1 score, a key aspect was the study of the built-in attention maps of Vision Transformer and post-interpretation of the performance of the trained ResNet50 model using the Grad-CAM method. The experiments were performed on the original dataset, as well as three of its modifications: with a zeroed background (threshold), with modified image areas using the inpainting method, and with a completely cleared background using zeroed background areas. To evaluate the generality of the attention mechanism in Vision Transformer, a test was also conducted on the classic MNIST handwritten digit recognition task. The results showed that the Vision Transformer architecture exhibits more localized and biologically based attention heatmaps, as well as greater resilience to changes in background noise.

Keywords: Vision Transformer; ResNet50; Grad-CAM; attention maps; attention heat maps; interpretability; classification; bacteria; image analysis.

For citation: Gridin V.N., Novikov I.A., Salem B.R., Solodovnikov V.I. Comparison of the interpretability of ResNet50 and ViT-224 models in the classification task is erroneous on images of a scanned microscope object. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 233-242 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-15.

Acknowledgements. This study was supported as part of topic no. № FFNR-2024-0003.

1. Введение

Одной из важнейших проблем современной медицинской науки является своевременное обнаружение патогенных микроорганизмов в исследуемом биоматериале, что крайне актуально при выборе тактики лечения осложнений вирусных инфекций, так как вызывающие эти осложнения бактериальные или грибковые возбудители могут давать схожую клиническую картину, а их медикаментозная терапия отличается диаметрально. Применение лантаноидного контрастирования в сочетании со сканирующей электронной микроскопией (СЭМ) позволяет получать серии снимков микробиологических объектов высокого пространственного разрешения, что увеличивает информативность визуальных данных за счет приобретения индивидуальных черт схожими микроорганизмами, у каждого из которых появляется характерный паттерн контрастирования. На сегодняшний день современные методы анализа изображений, включая локализацию, сегментацию и классификацию объектов, основаны на глубоких нейросетевых архитектурах. Для оценки функционирования обученных моделей традиционный акцент делается на метриках точности, полноты и F1-меры, однако они не отражают, каким образом модель принимает решения – на релевантных ли пикселях фокусируется сеть или извлекает корреляции с фоновыми артефактами и шумом [1].

С появлением и внедрением архитектуры Vision Transformer (ViT) с её механизмом самовнимания (self-attention) появилась возможность непосредственно визуально исследовать внутренние весовые матрицы – карты внимания (attention maps), и тем самым оценивать, какие зоны изображения оказывают наибольшее влияние на процесс классификации. Напротив, для популярных моделей сверточных нейронных сетей (Convolution Neural Network, CNN), таких как ResNet50, зачастую используют пост-интерпретационные методы визуализации с построением тепловых карт, в частности Grad-CAM (Gradient-weighted Class Activation Mapping), основанный на анализе градиентов по признакам последнего сверточного блока.

Цель проведенного исследования – провести сравнительный анализ интерпретируемости обученных моделей ViT-224 и ResNet50 при решении задачи классификации бактерий на изображениях, полученных посредством сканирующего электронного микроскопа, оценить их устойчивость к фоновой информации и расширить выводы через анализ современных методов интерпретации (R-Cut [5], SaCo [6], CDAM [7]). Помимо этого, для оценки правдоподобности результата тепловых карт дополнительно оценивается способность модели ViT к обобщению на классической задаче MNIST, что позволяет исключить специфику микробиологических снимков и проверить глобальную эффективность внимания модели.

2. Актуальность и обзор существующих методов

В статье [1] при решении задачи выявления пневмоторакса продемонстрировано превосходство карт внимания ViT над Grad-CAM CNN в плане фокусировки на патологии лёгких и соответствия экспертным оценкам радиологов. Авторы работы в BMC Medical Imaging представили ViT-модель для поиска туберкулёза, дополнив её методом Grad-CAM, и показали, что Grad-CAM-тепловые карты помогают локализовать очаги инфекции с точностью до 0.98 по метрике Accuracy [2]. В работе [3] приведен комбинированный подход Rollout + Grad-CAM, который позволяет улучшить согласованность карт внимания с аннотациями экспертов, что применимо в областях офтальмологии и рентгенологии. Также отмечено успешное использование модели ViT к двумерным биомедицинским данным, где карты внимания захватывали мелкие структуры с более высокой точностью, чем Grad-CAM для ResNet50 [4]. Метод R-Cut [5] предлагает фильтрацию фонового шума и усиление связей между значимыми блоками модели, что даёт более чистые и непрерывные карты внимания по сравнению с подходами Rollout и Grad-CAM. В работе [6] для оценки достоверности тепловых карт вводится метрика SaCo (Salience-guided Faithfulness Coefficient), что позволяет выявлять методы с высокоточными объяснениями (например, Transformer – Layer-wise Relevance Propagation (LRP)) по сравнению со случайными картами. Метод Class – Discriminative Attention Maps (CDAM) [7] комбинирует внимание и градиенты для получения класс-специфичных тепловых карт, что особенно полезно при мелких объектах и множественных экземплярах. В работе [9] показано, что комбинированные методы (механизм внимания + градиенты) наиболее стабильно указывают на значимые области даже после частичного или полного закрашивания ряда объектов на изображении. Авторы работы [10] продемонстрировали применение ViT и Score-CAM к классификации лейкоцитов, где внимание на морфологических особенностях клеток совпадало с экспертными аннотациями гематологов, а точность модели достигла 99.4 % [10].

Исследования показывают, что CNN часто полагаются на фоновые корреляции, тогда как ViT, благодаря глобальному механизму внимания, лучше справляется с шумами и артефактами, контрольные показатели ImageNet-C (-P) представлены в работе [8]. Эти результаты подчёркивают потенциал самовнимания для биомедицинской классификации мелких объектов.

3. Материалы и методы

3.1 Данные

Исходный набор изображений был получен с использованием электронного микроскопа Zeiss EVO 10 (Германия) с оригинальным детектором обратно-рассеянных электронов. Предварительная пробоподготовка осуществлялась с использованием химических реактивов для лантаноидного контрастирования микробиологических объектов BioREE-B (ООО «Глаукон», Россия). Набор данных включал 153 изображения с 9 классами бактерий (табл. 1), разделённый на обучающую (80 %) и тестовую (20 %) выборки. Изображения исходно имели разрешение 1024 × 720, были масштабированы до 224 × 224 и нормированы по среднему и стандартному отклонению ImageNet. Применялись базовые аугментации: случайные повороты (±15°), горизонтальные и вертикальные отражения, изменение яркости и контраста (±10 %).

Табл. 1. Декомпозиция исходного набора данных по штамму бактерий.
Table 1. Decomposition of the original dataset by bacterial strain.

Штамм бактерий	**Количество изображений**
A_baumannii_bl	17
E_coli	18
P_aerug	22
S_aureus	18
S_epid	20
Salm	21
Shig	22
Shig_sonnei	15

3.2 Глубокие нейросетевые модели

Для исследования использовались две архитектуры глубоких нейронных сетей для классификации изображений: ResNet-50 и Vision Transformer ViT-Base (ViT-Base/16, входной размер изображений 224×224).

Модель ResNet-50 – сверточная нейронная сеть с 50 слоями, включающая остаточные соединения. В данном эксперименте загружалась с предобученными на ImageNet весами и дообучалась на целевом наборе данных на протяжении 100 эпох. В процессе обучения использовался оптимизатор AdamW с начальными параметрами: скорость обучения lr=1e-5 и коэффициент регуляризации weight_decay=0.01.

Модель ViT-Base/16 (224×224) – трансформер-архитектура, обрабатывающая изображения с разбиением на изображения размером 16×16 пикселей и глубиной 12 слоев, загружалась с предобученными на ImageNet весами и дообучалась на целевом наборе данных в течение 50 эпох с использованием параметров AdamW при lr=1e-4 и weight_decay=0.05.

Для интерпретации результатов классификации применялись различные методы визуализации внимания. В случае ResNet-50 использовался метод Grad-CAM: тепловые карты формировались на основе градиентов последнего сверточного блока и нормировались в диапазон [0,1]. Для ViT-Base применялась техника Attention Rollout, при которой матрицы внимания трех последних слоев последовательно перемножались, а итоговая карта нормировалась аналогичным образом.

4. Результаты работы обученных моделей на исходном наборе изображений

Сравнение результатов функционирования обученных моделей представлены метриками точности, полноты и F1-меры в табл. 2. Обе модели продемонстрировали значения F1-меры более 0.9, что может привести к заключению о хорошей способности моделей обучаться на исходных изображениях с последующим верным прогнозом класса бактерий на тестовых изображениях. Стоит отметить, что ResNet50 продемонстрировал высокие значения полноты, а ViT в свою очередь значительно более высокие значения точности.

Табл. 2. Метрики качества обученных моделей ResNet50 и ViT при классификации бактерий.
Table 2. Quality metrics of trained ResNet50 and ViT models are excluded from classification.

Эксперимент	Точность	Полнота	F1-мера
ResNet50 на исходном наборе данных	0.89	0.975	0.929
ViT-224 на исходном наборе данных	0.95	0.895	0.912

На рис. 1, где демонстрируются тепловые карты с выделенными зонами с наибольшим влиянием на прогноз модели, можно заметить недостатки функционирования ResNet50 + GradCAM в виде обширных выделенных зон, не позволяющих в явном виде определить наиболее важные участки на каждом изображении. Механизм внимания в свою очередь демонстрируют более сконцентрированные участки, однако в обоих способах интерпретации работы моделей наблюдаются фоновые зоны с высокой значимостью на прогноз модели. Это может свидетельствовать о влиянии фонового шума на формирование прогноза, что в целом имеет негативное влияние на способность генерализации моделей.

5. Функционирование ViT модели на модифицированном наборе изображений

Для более подробного изучения работы механизма внимания было решено использовать архитектуру ViT как базовую модель, на основе которой осуществить оценку её функционирования при искажении исходного набора данных. В качестве возможных искажений были выбраны три различные вариации предобработки, которые позволили выделить зоны с бактериями или же наоборот их исключить для попытки выполнения прогноза только с использование фонового шума. Таким образом, исходный набор изображений подвергся следующим вариантам преобразований:

1. Порог (Threshold): адаптивная бинаризация по методу Otsu для обнуления слабых фоновых пикселей;
2. Маскирование (Inpainting): маскирование областей бактерий на основе порога интенсивности и восстановление фоновых областей алгоритмом Telea;
3. Cleaned (очистка фона): полное удаление сегментированных областей бактерий без дальнейшего восстановления.

Из рис. 2 видно, что при различных типах предобработки фокус модели значительно варьируется. В случае маскирования модель оценивает весь кластер бактерий на изображении. В случае подхода предобработки с порогом поведение модели схоже с оригинальным изображения, фокусируясь на определенных зонах с бактериями. Очистка фона вынуждает модель игнорировать пустые зоны бактерий, фокусируясь на фоновом шуме.

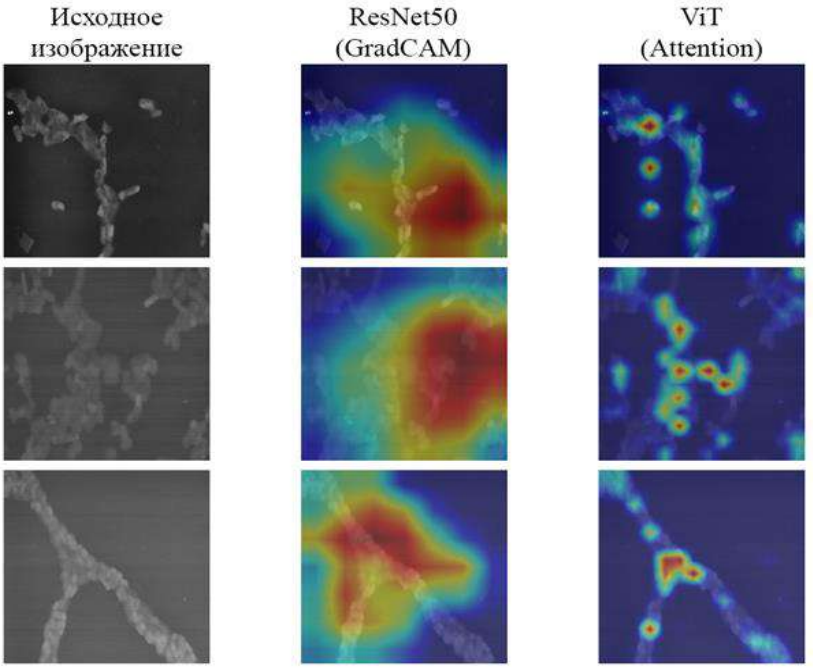


Рис. 1. Сравнение тепловых карт.
Fig. 1. Heat Map Comparison.

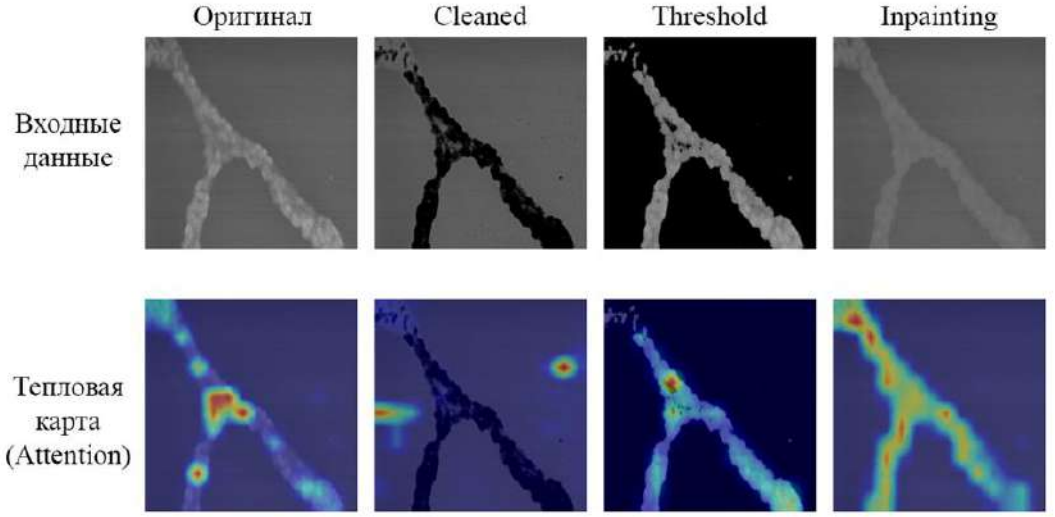


Рис. 2. Сравнение различных предобработок фона и соответствующих тепловых карт механизма внимания.
Fig. 2. Comparison of different background preprocessing and corresponding attention heatmaps.

Однако при сравнении метрик качества моделей, представленных в табл. 3, можно заметить, что все три модели имеют схожие результаты, причем наилучшие результаты у модели с удалением кластеров бактерий из изображения. Этот факт демонстрирует высокое влияние шума при корректном прогнозе, так как модель при отсутствии бактерий на изображении имеет F1-меру со значением 0.68 при классификации бактерий.

Табл. 3. Метрики модели ViT классификации бактерий при различных изменениях исходного набора данных.
Table 3. Metrics of the ViT model for classifying bacteria under various changes in the original dataset.

Эксперимент	Точность	Полнота	F1-мера
ViT-224 на исходном наборе данных	0.95	0.895	0.9115
ViT-224 (порог)	0.7464	0.635	0.640
ViT-224 (маскирование)	0.6833	0.6167	0.6346
ViT-224 (очистка фона)	0.7833	0.6479	0.6818

6. Классификация изображения на MNIST

Отдельно для проверки универсальности механизма внимания для ViT-224 был проведен эксперимент на наборе данных MNIST. Одноканальные изображения цифр масштабировались до 224 × 224 и дублировались по трём каналам для достижения цветовой схемы RGB. Модель дообучалась 20 эпох с параметрами batch_size=16 и lr=1e-4 без аугментаций. Динамика F1-score отслеживалась каждые 5 эпох, что позволило убедиться в стабильности поведения карт внимания вне специфики микробиологических изображений.

Табл. 4. Метрики модели ViT при классификации объектов из набора данных MNIST.
Table 4. ViT model metrics for classifying objects from the MNIST dataset.

Эксперимент	Точность	Полнота	F1-мера
ViT-224 на MNIST	0.964	0.961	0.961

На рис. 3 показаны примеры карт внимания для цифр «2», «7» и «0». Можно заметить, что модель локализует основные штрихи, определяющие цифру, и игнорирует фон (однородное поле).

Причем отдельные сегменты цифр на тепловых картах смещаются к ближайшим оставшимся пикселям контура, что подтверждает достоверность поведения тепловой карты и согласуется с метрикой SaCo [6].

7. Обсуждение

Сравнение двух подходов показало, что механизм самовнимания в ViT даёт более цельные и непрерывные тепловые карты, нежели фрагментированные градиентные тепловые карты Grad-CAM. Механизм самовнимания в ViT универсально выделяет ключевые структурные элементы объектов независимо от их масштаба и текстуры. Полученные изображения внимания на MNIST подтверждают, что информация из внутренних весов действительно отражает логику работы модели, а не случайные корреляции. Это укрепляет вывод о том, что результаты работы механизма внимания на микроскопических изображениях также являются достоверным отражением того, какие области влияют на итоговое предсказание.

Несмотря на схожие значения метрик качества, в частности F1-меры, на исходном наборе изображений анализ тепловых карт показывает: ViT-модель принимает решения, опираясь

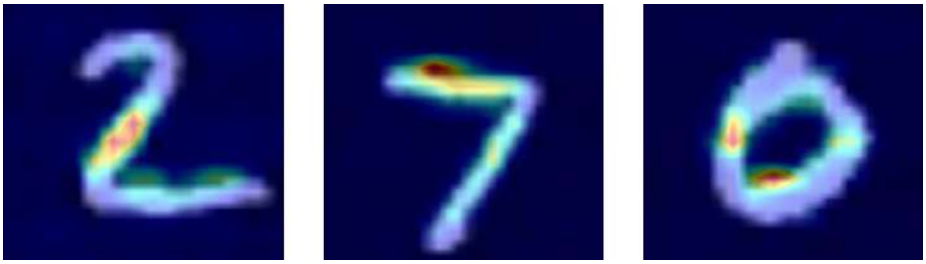


Рис. 3. Тепловые карты Attention поверх данных MNIST.
Fig. 3. Attention heatmaps on top of MNIST data.

преимущественно на данные о бактериях, тогда как ResNet50 учитывает более обширные зоны, включая фоновые артефакты. Эти результаты согласуются с работами Wollek et al. [1], Huang et al. [2] и Smith et al. [4], где ViT-модель концентрировала внимание на клинически значимых зонах.

Эксперименты с предобработкой порогом и маскированием подтвердили гипотезу о том, что ViT благодаря глобальному механизму внимания, устойчивее к удалению или модификации фона (ImageNet-C/P [8]), тогда как Grad-CAM для ResNet50 демонстрирует расфокусировку на оставшиеся текстуры.

8. Заключение

В ходе проведенного исследования был выполнен сравнительный анализ интерпретируемости двух ключевых архитектур глубокого обучения при решении задачи классификации бактерий на изображениях, полученных посредством сканирующего электронного микроскопа. Эксперименты на модифицированных наборах изображений и классическом MNIST подтвердили, что механизмы внимания в ViT-224 обеспечивают более локализованные и стабильные карты внимания. Полученные результаты имеют практическое значение для разработки интерпретируемых инструментов в микробиологии и клинической диагностике. В перспективе целесообразно расширить исследование на другие биомедицинские задачи, а также изучить влияние архитектурных изменений ViT на качество интерпретации.

Дополнение набора данных новыми представителями штаммов бактерий, а также введение контрольной группы (нормы) для подготовки модели для практического использования имеет наибольший приоритет для дальнейшего развития инструментария. Помимо этого, планируется расширение используемых инструментов для получение более достоверных и стабильных методов и метрик интерпретации, такие как R-Cut, SaCo и CDAM.

Список литературы / References

[1]. Wollek A., Graf R. et al. Attention-based Saliency Maps Improve Interpretability of Pneumothorax Classification. arXiv:2303.01871 (2023).
[2]. Huang X. et al. Enhanced tuberculosis detection using Vision Transformers and Grad-CAM. BMC Medical Imaging (2025).
[3]. Chen L. et al. MedViT: A robust vision transformer for generalized medical image analysis. Signal Processing: Image Communication, 105 (2023).
[4]. Smith J., Patel R. et al. Implementing vision transformer for classifying 2D biomedical images. Scientific Reports 14, 63094 (2024).
[5]. Huang Y. et al. R-Cut: Relationship Weighted Cut for Denoising ViT Attention Maps. MDPI Image Analysis (2024).
[6]. Wu Z. et al. SaCo: Salience-guided Faithfulness Coefficient for Evaluating Explanations. CVPR 2024.

[7]. Brocki M. et al. Class-Discriminative Attention Maps (CDAM) for Vision Transformers. ICLR 2024.
[8]. Xiao T. et al. Evaluating Robustness of Vision Transformers under Common Corruptions. AAAI 2022.
[9]. Badisa S. et al. Inpainting the Gaps: Framework for Evaluating Explainability under Occlusion. The CVF Open Access 2024.
[10]. Katar S., Yildirim A. Interpretable Classification of Leukocytes with ViT and Score-CAM. PMC 2023.

Информация об авторах / Information about authors

Владимир Николаевич ГРИДИН – доктор технических наук, профессор. Научный руководитель Центра информационных технологий в проектировании Российской академии наук. Область научных интересов: информационные технологии, искусственный интеллект, системы автоматизации проектирования, численно-аналитические методы.

Vladimir Nikolaevich GRIDIN – Dr. Sci. (Tech.), Prof., Scientific Director of the Design Information Technologies Center Russian Academy of Sciences. Research interests: information technology, artificial intelligence, and CAD systems, including analytical methods.

Иван Александрович НОВИКОВ – старший научный сотрудник Центра информационных технологий в проектировании Российской академии наук. Область научных интересов: информационные технологии, интеллектуальный анализ данных.

Ivan Aleksandrovich NOVIKOV – senior researcher at the Design Information Technologies Center Russian Academy of Sciences. Research interests: information technology, data mining.

Басим Раед САЛЕМ – научный сотрудник Центра информационных технологий в проектировании Российской академии наук. Область научных интересов: искусственный интеллект, системы поддержки принятия решений.

Basim Raed SALEM – researcher at the Design Information Technologies Center Russian Academy of Sciences. Research interests: artificial intelligence, decision support systems.

Владимир Игоревич СОЛОДОВНИКОВ – кандидат технических наук. Директор Центра информационных технологий в проектировании Российской академии наук. Область научных интересов: информационные технологии, методы машинного обучения и искусственного интеллекта применительно к самоорганизующимся системам поддержки принятия решений и автоматизированным средствам анализа данных.

Vladimir Igorevich SOLODOVNIKOV – Cand. Sci, (Tech.). Director of the Design Information Technologies Center Russian Academy of Sciences. Research interests: information technology, methods of machine learning and artificial intelligence as applied to self-organizing decision support systems and automated data analysis tools.