ТРУДЫ

ИНСТИТУТА СИСТЕМНОГО ПРОГРАММИРОВАНИЯ РАН

PROCEEDINGS OF THE INSTITUTE FOR SYSTEM PROGRAMMING OF THE RAS

ISSN Print 2079-8156
Том 35 Выпуск 3

ISSN Online 2220-6426
Volume 35 Issue 3

Институт системного
программирования
им. В.П. Иванникова РАН

Москва, 2023

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 областях 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:

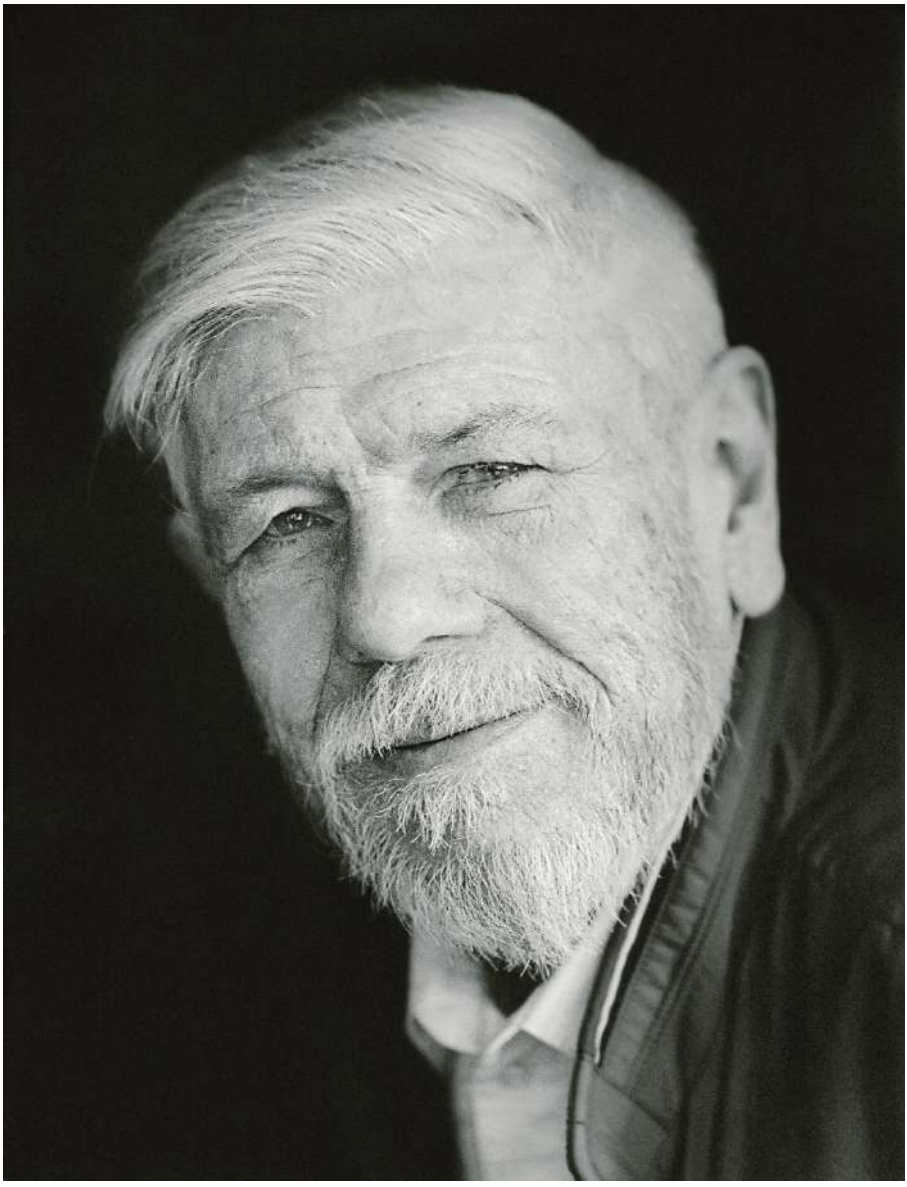
УДК004.45

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

Table of Contents



Сергей Дмитриевич Кузнецов
(08.04.1949 – 28.07.2023)

28 июля 2023 г. на 75-м году ушёл из жизни выдающийся учёный, бессменный заместитель главного редактора журнала «Труды Института системного программирования РАН» профессор, доктор технических наук Сергей Дмитриевич Кузнецов.

Сергей Дмитриевич Кузнецов родился 8 апреля 1949 года в Москве. В 1971 году он окончил механико-математический факультет МГУ им. М. В. Ломоносова по специальности «математик». После окончания МГУ работал в Институте точной механики и вычислительной техники им. С. А. Лебедева (ИТМ и ВТ), был одним из основных разработчиков операционной системы центрального процессора многомашинного вычислительного комплекса АС-6. Информационно-вычислительная система АС-6 активно использовалась в системах реального времени в центрах управления полётами космических аппаратов, участвовала в международной программе «Союз-Аполлон».

В 1980-х годах С. Д. Кузнецов, работая в НИИ «Дельта» Министерства электронной промышленности СССР и в Институте проблем кибернетики АН СССР, занимался созданием и внедрением программного обеспечения суперЭВМ «Электроника СС БИС-1», а также разработкой кластерной операционной системы (КЛОС) и Системы управления реляционной базы данных на базе КЛОС.

После образования Института системного программирования РАН в 1994 году С. Д. Кузнецов был его первым Ученым секретарем, осуществлял научное руководство отделом управления данными и разработки информационных систем, в задачи которого входила разработка системного программного обеспечения для обработки и анализа данных, системы управления базами данных, технологии распределенной обработки больших данных, технологии облачных вычислений. Кроме того, под его научным руководством в ИСП РАН начиналась разработка алгоритмов статистического анализа данных и машинного обучения, а также программного обеспечения для решения прикладных задач, в том числе для интеллектуального анализа текстов, анализа социальных сетей, задач биоинформатики и обработки мультимедийных данных.

В 1979 году С. Д. Кузнецов защитил кандидатскую диссертацию по организации мультипрограммирования в операционной системе центрального процессора АС-6, в 1994 году защитил докторскую диссертацию по созданию открытого SQL-сервера. Под его руководством впервые в нашей стране была создана XML-СУБД Sedna.

С 1989 по1995 год С. Д. Кузнецов вел большую общественно-научную работу, Будучи одним из первых специалистов в стране, которые начали активно изучать и использовать операционную систему Unix, он стал первым председателем Совета Советской, а потом Российской ассоциации пользователей ОС Unix (SUUG), членом Европейской ассоциации EurOpen, ассоциаций Usenix и Uniforum, членом ACM и ACM SIGMOD, членом IEEE Computer Society, представителем IEEE Computer Society в Москве, заместителем председателя Московской секции ACM SIGMOD, членом программных комитетов международных конференций DEXA, SOFSEM, ADBIS, ISD, BulticDB, SYRCoDIS, DAMDID.

С. Д. Кузнецов был первым главным редактором журнала «Открытые системы», научным редактором журнала «СУБД». До последнего времени был редактором тематической рубрики журнала «Открытые системы», членом редколлегии журнала «Вычислительные методы и программирование».

С. Д. Кузнецов много сил и энергии отдавал воспитанию и обучению молодых специалистов на кафедрах МГУ, МФТИ и ВШЭ. Его общий научно-педагогический стаж превысил 50 лет. Под его руководством подготовлен большой коллектив высококвалифицированных учёных и разработчиков программного обеспечения вычислительных систем. С. Д. Кузнецов автор более 200 научных работ и четырех учебников по СУБД.

Высокий уровень публикаций в журнале «Труды Института системного программирования РАН», подтверждаемый постоянным ростом рейтинга нашего журнала, во многом является результатом многолетних усилий С. Д. Кузнецова и его дружелюбной манеры общения с авторами, рецензентами и членами редакционной коллегии.

Уход С. Д. Кузнецова – это невосполнимая утрата, многим из нас он был испытанным другом, многим учителем и наставником. Память о Сергее Дмитриевиче Кузнецове будет жить в сердцах его учеников, коллег, авторов и рецензентов нашего журнала и всех сотрудников ИСП РАН.

Редакционная коллегия журнала
«Труды Института системного программирования РАН»

DOI: 10.15514/ISPRAS-2023-35(3)-1

Discovering Process Models from Event Logs of Multi-Agent Systems Using Event Relations

A.A. Sherstyugina, ORCID: 0009-0009-2878-3565 <*aasherstyugina@edu.hse.ru*>
R.A. Nesterov, ORCID: 0000-0002-4162-9070 <*rnesterov@hse.ru*>

HSE University,
11 Pokrovsky boulevard, Moscow, 101000, Russia

Abstract. The structure of a process model directly discovered from an event log of a multi-agent system often does not reflect the behavior of individual agents and their interactions. We suggest analyzing the relations between events in an event log to localize actions executed by different agents and involved in their asynchronous interaction. Then, a process model of a multi-agent system is composed from individual agent models between which we add channels to model the asynchronous message exchange. We consider agent interaction within the acyclic and cyclic behavior of different agents. We develop an algorithm that supports the analysis of event relations between different interacting agents and study its correctness. Experimental results demonstrate the overall improvement in the quality of process models discovered by the proposed approach in comparison to monolithic models discovered directly from event logs of multiagent systems.

Keywords: Multi-agent systems; event logs; process discovery; Petri nets; event relations; asynchronous interaction.

For citation: Sherstyugina A.A., Nesterov R.A. Discovering Process Models from Event Logs of Multi-Agent Systems Using Event Relations. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 3, 2023. pp. 11-32. DOI: 10.15514/ISPRAS-2023-35(3)-1

Acknowledgments: This work is supported by the Basic Research Program at the HSE University, Russia.

Синтез моделей процессов по журналам событий мультиагентных систем с помощью отношений между событиями

А.А. Шерстюгина, ORCID: 0009-0009-2878-3565 <*aasherstyugina@edu.hse.ru*>
Р.А. Нестеров, ORCID: 0000-0002-4162-9070 <rnesterov@hse.ru>

Национальный исследовательский университет «Высшая школа экономики»,
Россия, 101000, Москва, Покровский бульвар, 11

Abstract. Структура модели процесса, синтезированной напрямую по журналу событий мультиагентной системы часто не дает представления о поведении отдельных агентов, а также о способе их взаимодействия. Для локализации действий, которые выполняются различными агентами и которые вовлечены в их асинхронное взаимодействие мы выделяем и анализируем отношения между событиями в журнале. В результате модель мультиагентной системы представляет собой композицию моделей поведения отдельных агентов, между которыми добавляются каналы асинхронного обмена сообщениями. В статье рассматривается как ациклическое, так и циклическое взаимодействие агентов. Нами предложен и обоснован алгоритм выделения и анализа отношений между событиями в журнале событий мультиагентной системы. Результаты экспериментальной оценки разработанного алгоритма

подтверждают общее улучшение качественных оценок моделей процессов, синтезированных по журналам событий мультиагентных систем с помощью отношений между событиями в сравнении с монолитными моделями, которые синтезируются напрямую.

Ключевые слова: Мультиагентные системы; журналы событий; синтез моделей процессов; сети Петри; отношения между событиями; асинхронное взаимодействие.

Для цитирования: Шерстюгина А.А., Нестеров Р.А. Синтез моделей процессов по журналам событий мультиагентных систем с помощью отношений между событиями. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 11–32 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–1

Благодарности: Работа поддержана Программой фундаментальных исследований Национального исследовательского университета «Высшая школа экономики» (НИУ ВШЭ), Россия.

1. Introduction

The behavior of an information system is frequently recorded in *event logs*. They can register, for instance, user activities, transaction executions, or message exchanges. An event log consists of finite sequences (traces) of events ordered by the occurrence time. *Process mining* uses event logs to *discover* models reflecting the actual state of processes in an information system. Process models discovered from event logs capture considerable changes that can be introduced to an information system during its operation, while models manually created at the initial life-cycle stages do not take these changes into account [1].

A record in a trace of an event log usually includes not only the identifier of an action, but also other attributes, which can specify the resources necessary for executing the recorded action. These attributes can also designate who executes an action. For example, Table 1 shows a trace of an event log, where an action record has the «Agent» attribute, and actions are executed by two agents: *Peter* or *Alex*. We say that an event log where actions are attributed with the information on agents records the behavior of a *multi-agent system*.

Process models can be discovered in a variety of notations, including different classes of Petri nets, transition systems, and BPMN (Business Process Model and Notation). In our paper, we focus on modeling the *control-flow* of processes, i.e., the causal dependencies among events in a log. Thus, we will apply *Petri nets* [2] — the formalism extensively used to model and analyze the properties of process behavior.

Table 1. A trace in an event log of a multi-agent system

Timestamp	Action	Agent
30-12-2022:14.45	prepare msg	Peter
05-01-2023:09.34	send msg	Peter
07-01-2023:12.12	receive msg	Alex
12-01-2023:13.25	send ack	Alex
12-01-2023:14.55	receive ack	Peter
12-01-2023:14.55	local check	Alex

Petri nets are also a convenient tool to model the interaction between different components in a multi-agent system. Fig. 1 shows two Petri nets N_1 and N_2 representing two agents with the sequential behavior. They exchange messages through two distinguished channel nodes *a* and *b*.

Recent papers in the field of process mining also demonstrate the shift in a focus to a discovery of process models with an understandable structure reflecting the complex synchronizations between objects [3], the hierarchy of activities [4, 5], or the interaction-oriented viewpoints of the architecture of a multi-agent system [6].

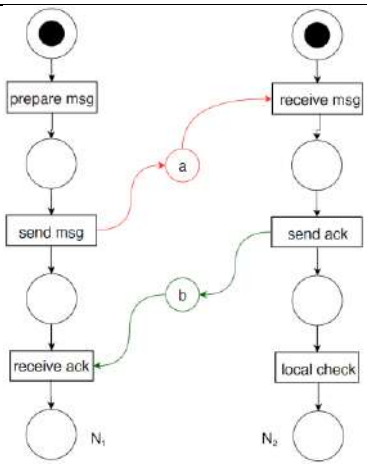


Fig. 1. A multi-agent system with two asynchronously interacting agents

The paper [6] proposed a *compositional* approach to discovering an *architecture-aware* process model from an event log of a multi-agent system. The structure of an architecture-aware process model explicitly reflects agent behavior and their interactions similar to *Fig. 1*, where two agents exchange message through channels `a` and `b`. A model is constructed by a composition of individual agent models controlled by a manually selected interface pattern model. An *interface pattern* provides a high-level specification of agent interactions. However, in the case of the poor selection of an interface model, one has to reconfigure it and perform an additional check of a reconfigured model.

Here, we propose to ease this restriction on making the preliminary choice of an interface pattern. We suggest to identify asynchronous agent interactions using causal relations between events extracted directly from an event log of a multi-agent system. For instance, in an event log obtained by simulating a process model shown in Fig. 1 the occurrence of *"send msg"* action will always be recorded before the occurrence of *"receive msg"* action. Extracting such causality relations will help us to localize events in a log corresponding to the occurrence of actions executed by different agents and involved in their asynchronous communication. Correspondingly, we will determine transitions in individual agent models to be connected via an asynchronous channel.

Note that the automated discovery of process models from event logs is supported by a wide range of algorithms [7]. They usually deal with typical problem of event data representation, including, for instance, *noise* (missing or duplicated records) and *incompleteness*, i.e., a finite event cannot cover all possible process executions. The paper [6] also stressed that an event log of a multi-agent system requires the additional inspection of agent behavior, since the direct discovery from a multi-agent system event log produces process models the structure of which does not explicitly reflect agent behavior as sub-models and agent interactions as distinguished nodes. This happens because the concurrent execution of relatively independent agents leads to a wide range of possible traces recorded in an event log of a multi-agent system.

The quality of discovered process models is the main subject in *conformance checking* [8], which proposes a collection of different dimensions to evaluate the correspondence between an event log and a process model. Fitness and precision are two widely-used quality metrics that can characterize a discovered process model. *Fitness* is an estimation of the ratio of the traces executable by the model to the total number of traces in an event log. A model with the perfect fitness can execute every trace in an event log. For example, the model shown in Fig. 1 can execute the trace in Table 1, if we consider N_1 as the behavior of *Peter*, and N_2 as the behavior of *Alex*. *Precision* evaluates the ratio of the behavior recorded in an event log and the behavior allowed by a process model. A process model with the perfect precision can only execute traces in an initial event log. The perfect precision

limits the use of a discovered process model, since any event log of an information system represents only a finite "snapshot" of all possible process executions.

An architecture-aware process model discovered from an event log of a multi-agent system using the compositional approach of [6] is guaranteed to possess the perfect fitness. The approach to the analysis of agent interactions using causal event relations in a log, proposed in our study, will also ensure the perfect fitness of the process model of a multi-agent system obtained by connecting individual agent models via asynchronous channels. The main results presented in this paper are:

1) An approach to the analysis of causality relations between events in an event log of a multi-agent system for the identification of specific events involved in the asynchronous communication between different agents.
2) Demonstration of the approach correctness and its experimental evaluation.

The remainder of this paper is organized as follows. In the next section, we collect the formal background of our approach to the analysis of event relations in an event log, including generalized workflow nets (GWF-nets) — a class of Petri nets used to model the behavior of agents and multi-agent systems. Section 3 considers the localization of events in an event log corresponding the asynchronous agent interactions within the acyclic agent behavior. Section 4 explores the case of localizing asynchronous interactions among agents with cycles. Section 5 reports the outcomes from the experimental evaluation. In Section 6, we review the related research, and Section 7 concludes the paper.

2. Background

In this section, we aim to provide the basic definitions concerning several general notions, event logs, and generalized workflow nets. We refer to these definitions when describing our approach to the analysis of causal event relations involving different agents.

S^+ denotes the set of all finite non-empty sequences over a finite set S, and $S^* = S^+ \cup \{\varepsilon\}$, where ε is the *empty* sequence. Let $\sigma \in S^*$ and S' be a subset of S. Then $\sigma|_{S'}$ denotes the projection of σ on S'. In other words, $\sigma|_{S'}$ is the subsequence of σ obtained by removing elements not belonging to S'. For example, let $S = \{a, b, c, d\}, \sigma = abadabcdcb \in S^*$, and $S' = \{b, c\}$. Projecting σ on S' gives $\sigma|_{S'} = bbcb$. If $s \in S$ occurs in a sequence $\sigma \in S^*$, then we write $s \in \sigma$.

$\mathbb{N}$ denotes the set of non-negative integers. A function $m: S \to \mathbb{N}$ defines a *multiset* m over a non-empty set S. We write $s \in m$ iff $m(s) > 0$. The set of all finite multisets over S is denoted by $\mathcal{B}(S)$. Let $m_1, m_2 \in \mathcal{B}(S)$. Then $m_1 \subseteq m_2$ iff $m_1(s) \leq m_2(s)$; $m' = m_1 \cup m_2$ if $m'(s) = m_1(s) + m_2(s)$; $m'' = m_1 \setminus m_2$ iff $m''(s) = \max(m_1(s) - m_2(s), 0)$ for all $s \in S$.

2.1 Event Logs

An event log is the main input to a process discovery algorithm. It contains a multiset of *traces* — ordered event sequences.

Definition 1 (Event log). Let $\mathcal{A}$ denote the set of actions. A *trace* σ is a finite non-empty sequence over $\mathcal{A}$, i.e., $\sigma \in \mathcal{A}^+$. An event log L is a multiset of traces over $\mathcal{A}$, i.e., $L \in \mathcal{B}(\mathcal{A})$.

When we consider an event log of a multi-agent system with two asynchronously interacting agents, the set $\mathcal{A}$ can be partitioned into two disjoint subsets, i.e., $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, s.t. $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$, where $\mathcal{A}_1$ ($\mathcal{A}_2$) is the set of actions executed only by the first (second) agent.

To discover an individual model of a multi-agent system, we need to project all traces in L onto the set of actions executed by the corresponding agent. The projection of an event log over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ on $\mathcal{A}_1$ is denoted by $L_{\mathcal{A}_1}$. Constructing $L_{\mathcal{A}_1}$ requires projecting every trace $\sigma \in L$ on $L_{\mathcal{A}_1}$, i.e., taking $\sigma|_{\mathcal{A}_1}$. We take into account only non-empty projections $\sigma|_{\mathcal{A}_1}$ and pay additional attention to coinciding projections.

For example, a trace shown in Table 1 can be projected onto the set of action executed only by Peter or by Alex.

Let us consider basic causality relations between events recorded in a log L over $\mathcal{A}$, which are determined by the order of corresponding records in the traces of L. Thus, two events $a_1, a_2 \in \mathcal{A}$ are:

1) in the *precedence* relation (a_1 precedes a_2), denoted $a_1 < a_2$, iff $\forall \sigma \in L$: if $a_1, a_2 \in \sigma$, then $\sigma = \sigma' a_1 \sigma'' a_2 \sigma'''$, where $\sigma', \sigma'', \sigma''' \in (\mathcal{A} \setminus \{a_1, a_2\})^*$;
2) in the *following* relation (a_1 follows a_2), denoted $a_1 > a_2$, iff $\forall \sigma \in L$: if $a_1, a_2 \in \sigma$, then $\sigma = \sigma' a_2 \sigma'' a_1 \sigma'''$, where $\sigma', \sigma'', \sigma''' \in (\mathcal{A} \setminus \{a_1, a_2\})^*$;
3) in the *parallel* relation (a_1 is in parallel with a_2), denoted $a_1 >< a_2$, if there exists a trace $\sigma \in L$, s.t. $\sigma = \sigma' a_1 \sigma'' a_2 \sigma'''$, and a trace $w \in L$, s.t. $w = w' a_1 w'' a_2 w'''$, where $\sigma', \sigma'', \sigma''', w', w'', w''' \in (\mathcal{A} \setminus \{a_1, a_2\})^*$.

It follows that the precedence and the following relations are transitive. For example, $a_1 < a_2$ and $a_2 < a_3$ together leads to traces of the form $\sigma = \ldots a_1 \ldots a_2 \ldots a_3 \ldots$, which implies $a_1 < a_3$. If required by the context, we can also use the $<_L$ relation sign to explicitly show to which event log this relation corresponds.

2.2 Generalized Workflow Nets

Workflow nets (WF-nets) [9] are among basic process models discovered from event logs. A WF-net is a special class of a Petri net with the distinguished initial and final places. The execution of a trace in an event log directly corresponds to the execution of a WF-net from its initial to its final place. We will use generalized workflow nets (GWF-nets), as in [6], to model the behavior of agents and multi-agent systems. Here, we define GWF-nets and their behavior.

Definition 2 (Net). A net is a triple $N = (P, T, F)$, where P and T are two disjoint sets of places and transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation. For any node $x \in P \cup T$:

1) $\bullet x = \{y \in P \cup T \mid (y, x) \in \mathrm{F}\}$ is the *preset* of x.
2) $x \bullet = \{y \in P \cup T \mid (y, x) \in \mathrm{F}\}$ is the *postset* of x.
3) $\bullet x \bullet = \bullet x \cup x \bullet$ is the *neighborhood* of x.

In our study, we consider nets without self-loops, i.e., $\forall x \in P \cup T: \bullet x \cap x \bullet = \emptyset$ and isolated transitions, i.e., $\forall t \in T: |\bullet t| \geq 1$ and $|t \bullet| \geq 1$.

The $\bullet$-notation is also extended to subsets of nodes. Let $N = (P, T, F)$ be a net, and $Y \subseteq P \cup T$. Then $\bullet Y = \bigcup_{y \in Y} \bullet y$, $Y \bullet = \bigcup_{y \in Y} y \bullet$ and $\bullet Y \bullet = \bullet Y \cup Y \bullet$. $N(Y)$ denotes the subnet of N *generated* by Y, i.e., $N(Y) = \big(P \cap Y, T \cap Y, F \cap (Y \times Y)\big)$.

Let $N = (P, T, F)$ be a net, and $t_1, t_2 \in T$. Transitions t_1, t_2 are in *conflict* iff $\bullet t_1 \cap \bullet t_2 \neq \emptyset$. N is *conflict-free* if no transitions are in conflict.

A marking (state) m in a net $N = (P, T, F)$ is a multiset over P, i.e., $m: P \to \mathbb{N}$. Marking is safe iff $\forall p \in P: m(p) \leq 1$, i.e., a safe marking is a set of places. Marking m of place $p \in P$ is depicted by putting $m(p)$ black dots inside p.

Definition 3 (Net system). A net system is a quadruple $N = (P, T, F, m_0)$, where (P, T, F) is a net, and $m_0: P \to \mathbb{N}$ is the *initial* marking.

A marking m in a net $N = (P, T, F)$ enables transition $t \in T$, denoted $m[t\rangle$, iff $\bullet t \in m$. Enabled transitions may *fire*. Firing t at m evolves N to a new marking $m' = (m \setminus \bullet t) \cup t \bullet$, denoted $m[t\rangle m'$. A sequence $w \in T^*$ is a *firing sequence* in a net system $N = (P, T, F, m_0)$ if $w = t_1 t_2 \ldots t_n$ and $m_0[t_1\rangle m_1[t_2\rangle \ldots m_{n-1}[t_n\rangle m_n$. Then we write $m_0[w\rangle m_n$. The set of all firing sequences in N is denoted by $FS(N)$.

A marking m in $N = (P, T, F, m_0)$ is reachable if $\exists w \in FS(N): m_0[w\rangle m$. Any marking can be reached from itself by firing the empty sequence $m_0[\varepsilon\rangle m$. The set of all markings reachable from m is denoted by $[m\rangle$. N is safe iff all reachable markings in N are safe.

A *state machine* is a connected net (P, T, F), where$\forall t \in T: |\bullet\, t\,| = |t\, \bullet| = 1$. A subnet of $N = (P, T, F)$ generated by $Y \subseteq P$ and $\bullet\, Y\, \bullet$, i.e., $N(Y \cup \bullet\, Y\, \bullet)$, is a *sequential component* of N if it is a state machine and has a single token in the initial marking. N is covered by sequential components if every place belongs to at least one sequential component. In this case, N is *state machine decomposable* (SMD).

State machine decomposability is a basic feature bridging structural and behavioral properties of nets, also considered in [9] as an important feature of workflow nets. It is easy to see that SMD net systems are safe since their initial markings are safe. We further work with SMD net systems, unless otherwise stated explicitly. Thus, we omit SMD in their descriptions.

In a GWF-net, we impose additional restrictions on its initial marking (no arcs incoming to corresponding places) and distinguish its final marking (places without outgoing arcs). Compared to a classical WF-net, initial and final marking in a GWF-net can be sets of places rather than singletons.

Definition 4 (GWF-net). A generalized workflow net is a net system $N = (P, T, F, m_0)$ equipped with the final marking $m_f \subseteq P$ such that:

1) $\bullet\, m_0 = \emptyset$.
2) $m_f \bullet = \emptyset$.
3) $\forall x \in P \cup T\ \exists s \in m_0\ \exists f \in m_f: (s, x), (x, f) \in F^{RT}$, where F^{RT} is the reflexive transitive closure of F.

According to the third requirement in Definition 4, any node in a GWF-net lies on a path from a place in its initial marking to a place in its final marking. For instance, the Petri net shown earlier in Fig. 1 is a GWF-net, while the behavior of agents N_1 and N_2 can be considered as classical WF-nets with the single initial and final places.

3. Localizing Acyclic Agent Interactions

Here we discuss our approach to finding pairs of actions in an event log representing sending and receiving operations executed by *different* agents. Given an event log of a multi-agent system, we construct a matrix representation of event relations. Then we show how to identify the candidate pairs of events that may represent the asynchronous communication of different agents and connect corresponding transitions in the individual agent models.

3.1 Matrix Representation of Event Relations

Matrix representation of relations among events recorded in an event log facilitate the pair-wise analysis of events. For what follows, we consider the basic case of a multi-agent system with the *sequential* agent behavior, s.t., actions executed by a specific agent are recorded in an event log only in the precedence or in the following relation. We also show how our reasoning can be extended to agents with parallel and alternative behavioral constructs.

Let L be an event log over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, s.t. $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$. Correspondingly, $\mathcal{A}_1$ and $\mathcal{A}_2$ are two disjoint sets of actions executed by two asynchronously interacting agents. Assume $|\mathcal{A}_1| = m$ and $|\mathcal{A}_2| = n$.

We construct matrix R^L of size $m \times n$, which stores relations between the pairs of events representing the occurrence of actions executed by different agents. Given $a_i^1 \in \mathcal{A}_1$ and $a_j^2 \in \mathcal{A}_2$ with $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$, every element $r_{i,j}$ in R^L is defined by the following cases:

1) $r_{i,j} =$ " $<$ " iff $a_i^1 <_L a_j^2$;

2) $r_{i,j} = " > "$ iff $a_i^1 >_L a_j^2$;

3) $r_{i,j} = " >< "$ iff $a_i^1 ><_L a_j^2$.

Thus, event relations extracted from an event log L fully determines the values of the elements in the corresponding matrix R^L.

Figure 2 shows the example of a matrix representation for event relations constructed from an event log a multi-agent system with asynchronously interacting agents, where the first agents executes actions from the set $\mathcal{A}_1 = \{a_0, a_1, a_2\}$, and the second agent executes actions from the set $\mathcal{A}_2 = \{b_0, b_1, b_2, b_3\}$. For the convenience of the representation, we use names of actions instead of the indices of rows and columns. This matrix says that, for example, in all traces of the initial event log L, actions b_1 and a_2 are executed concurrently (independently), while action a_1 always precedes action b_1.

In addition, recall that agent behavior is considered to be conflict-free and sequential. Then we can easily order actions executed by the same agent according to the event relations, i.e., using the precedence relation. For instance, in *Fig. 2*, we have that $a_0 < a_1 < a_2$ and $b_0 < b_1 < b_2 < b_3$. This ordering of actions is done before constructing a matrix of event relations. It will help us simplify the further processing and identification of events representing the occurrence of sending-receiving operations between two agents.

	b_0	b_1	b_2	b_3
a_0	$><$	$<$	$<$	$<$
a_1	$><$	$<$	$<$	$<$
a_2	$>$	$><$	$><$	$><$

Fig. 2. A matrix of event relations between two asynchronously interacting agents

The intuition behind the asynchronous message exchange is rather straightforward. After putting a message to a channel, an agent can freely continue its job, while the other agent expecting to receive a message cannot continue to operate until the message is delivered.

This reasoning can also be shifted to our matrix representation of event relations. In a matrix of event relations constructed out of an event log of a multi-agent system with two sequential asynchronously interacting agents, we will be able to locate a “rectangle” formed by the adjacent rows and columns filled by the same event relation $" < "$ or $" > "$. This is justified by the fact that in all traces of an initial event log several events corresponding to the actions executed by the agent receiving a message are recorded strictly after several events corresponding to the actions executed by the agent who sends a message. Rectangular sections in an event relation matrix filled by the same precedence or following relation are called *regions*.

Definition 5. Let L be an event log over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, s.t. $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$, $|\mathcal{A}_1| = m$, $|\mathcal{A}_2| = n$. Let R^L be an event relation matrix constructed as described above. A rectangular section in R^L formed by k adjacent rows $i, i+1, \ldots, i+k-1$ and by ℓ adjacent columns $j, j+1, \ldots, j+\ell-1$ is a *p-region* (*f-region*) of R^L if and only if for all $i' = i, i+1, \ldots, i+k-1$ and $j' = j, j+1, \ldots, j+\ell-1$ we have that $r_{i',j'} = " < "$ ($r_{i',j'} = " > "$).

The region in an event relation matrix R^L starting from row a, column c and finishing at row b and at column d is briefly denoted by $R^L(a-b, c-d)$.

Note that we do not consider a region which is included in another one. We are looking for *maximal* regions in an event relation matrix. For instance, in the event relation matrix shown in Fig. 2, region $R^L(a_2 - a_2, b_0 - b_0)$, since it cannot be extended with other adjacent rows and columns, while $R^L(a_0 - a_1, b_1 - b_2)$ is not maximal, since it is a part of the bigger region $R^L(a_0 - a_1, b_1 - b_3)$ that is indeed maximal.

Further, while analyzing regions in an event relation matrix, we always consider maximal regions that cannot be extended with more adjacent rows and columns.

Let us take a closer look at the p-region $R^L(a_0 - a_1, b_1 - b_3)$ in the event relation matrix shown in Fig. 2. The occurrences of actions a_0 and a_1 were recorded before the occurrences of actions b_1, b_2 and b_3 in an event log L. Taking into account the sequential agent behavior, i.e., $a_0 < a_1 < a_2$ and $b_0 < b_1 < b_2 < b_3$, we can easily simplify three event relations $a_0 < b_1$, $a_0 < b_2$ and $a_0 < b_3$ to the single relation $a_0 < b_1$, which automatically ensures the remaining two relations. By analogy, three relations $a_1 < b_1$, $a_1 < b_2$ and $a_1 < b_3$ are simplified to $a_1 < b_1$. Finally, two relations $a_0 < b_1$ and $a_1 < b_1$ with $a_0 < a_1$ give us the single event relation $a_1 < b_1$.

Thus, the p-region $R^L(a_0 - a_1, b_1 - b_3)$ in the event relation matrix from Fig. 2 can be fully described by the single event relation a1 < b1 — the lower left corner of the corresponding rectangular area in the event relation matrix.

Event relation that fully describes a region in an event relation matrix is called the minimum of a region, i.e., other event relations within this region coincides with the minimum. It is easy to see that, if the minimum of a p-region is its lower left corner, then the minimum of an f-region is its upper right corner, as illustrated in *Fig. 3*, where the minimum is highlighted in red.

The minimum event relation in a region is the pair of events which can represent the occurrence of actions agents use for the asynchronous communication.

	...	b_j	...	$b_{j+\ell-1}$
...				
a_i		$<$	...	$<$
...		$<$	...	$<$
a_{i+k-1}		$<$	...	$<$
...				

	...	b_j	...	$b_{j+\ell-1}$
...				
a_i		$>$	...	$>$
...		$>$	...	$>$
a_{i+k-1}		$>$	...	$>$
...				

Fig. 3. Localizing minimum in a region of an event relation matrix

For example, the event relation matrix R^L shown in Fig. 2 has the p-region $R^L(a_0 - a_1, b_1 - b_3)$ with the minimum relation $a_1 < b_1$ and the f-region $R^L(a_2 - a_2, b_0 - b_0)$ with the minimum relation $a_2 > b_0$. The sequential behavior of corresponding agents can be easily represented via a Petri net with consequent transitions (see N_1 and N_2 in Fig. 4).

According to the minimal event relation of region in the event relation matrix R^L from Fig.2, we introduce two channel places between transitions a_1, b_1 (green place) and transitions b_0, a_2 (red place). Arcs connecting these places with transitions in Fig. 4 follow the direction of the corresponding minimum event relation.

In the following paragraph, we propose an algorithm, which identifies regions in the event relation matrix and finds their corresponding minimal event relations. We prove the algorithm correctness from the point of view of preserving the perfect fitness. We also show that there can be redundant minimum event relations representing different overlapping regions.

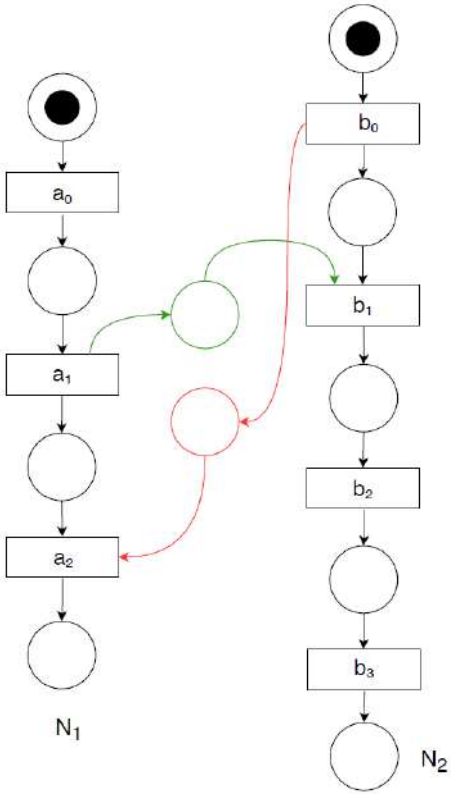


Fig. 4. Introducing channel places according to the matrix from Fig. 2

3.2 Algorithm for Finding Minimal Event Relations in Regions of an Event Relation Matrix

We start with an event log L over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ of a multi-agent system with two asynchronously interacting agents. Let $|\mathcal{A}_1| = m$ and $|\mathcal{A}_2| = n$. To simplify the processing of traces in L, we will construct a square event relation matrix R_0^L of size $(m + n) \times (m + n)$ storing event relations between all possible pairs of events in $\mathcal{A}$. The indices of an element $r_{i,j}^0$ in R_0^L will directly correspond the indices of actions a_i and a_j in $\mathcal{A}$. Afterwards, choosing necessary rows and columns in a square R_0^L representing the behavior of different agents, we will be able to easily form a required event relation matrix R^L, as described in the previous paragraph.

Here, instead of directly using relation signs, we will assign numbers: -1 for $<$ (precedence), 1 for $>$ (following), and 0 for $><$ (parallel). Initially, R_0^L is filled by the ordering of indices, where $i, j = 1, 2, \ldots, m + n$: (a) if $i < j$, then $r_{i,j}^0 = -1$; (b) if $i > j$, then $r_{i,j}^0 = 1$. We do not care about the values in R_0^L at its main diagonal (for $r_{i,i}^0$), since we do not consider the reflexive event relations.

Subsequently, we update R_0^L according to the actual relations between event pairs in L. Algorithm 1 shows how we process traces in L to extract corresponding event relations. Given a trace σ in an event log L, we consider every pair of two events preceding each other in σ and update $r_{i,j}^0$ to 0 only if it was 1 before, taking into account that actions executed by different agents are also sorted by the preceding relation. This intuitively means that we have the pair of events recorded in both following and precedence relation in a log representing the sequentialization of parallel execution.

Algorithm 1: Populating an event relation matrix

Input: L – an event log over $\mathcal{A} = \{a_1, a_2, ..., a_{m+n}\}$, R_0^L – an initial square even relation matrix

Output: R_0^L, where $r_{i,j}^0 = -1$ if $a_i <_L a_j$; $r_{i,j}^0 = 0$ if $a_i >\!<_L a_j$; $r_{i,j}^0 = 1$ if $a_i >_L a_j$

```
foreach σ ∈ L do
    foreach a_i, a_j ∈ A, s.t. σ = σ'a_iσ''a_jσ''' do
        if r^0_{i,j} = −1 or r^0_{i,j} = 0 then
            continue
        end
        if r^0_{i,j} = 1 then
            r^0_{i,j} = 0
        end
    end
end
```

For instance, Fig. 5 shows the square event relation matrix R_0^L, built according to Algorithm 1, corresponding to the earlier discussed R^L (see Fig. 2). The main diagonal in this R_0^L is filled with asterisk signs, since we ignore reflexive relations.

We filled two areas in this square matrix with different colors to demonstrate two possible ways of choosing rows and columns for further analysis of event relations corresponding to the occurrence of actions executed by different agents. It is also easy to refine the notion of a region w.r.t. the numerical representation of event relations.

	a_0	a_1	a_2	b_0	b_1	b_2	b_3
a_0	$*$	-1	-1	0	-1	-1	-1
a_1	1	$*$	-1	0	-1	-1	-1
a_2	1	1	$*$	1	0	0	0
b_0	0	0	-1	$*$	-1	-1	-1
b_1	1	1	0	1	$*$	-1	-1
b_2	1	1	0	1	1	$*$	-1
b_3	1	1	0	1	1	1	$*$

Fig. 5. A square matrix of event relations constructed by Algorithm 1

The p-region is the rectangular area of the numerical event relation matrix filled completely with -1, while the f-region should be filled only with 1. Here, we also consider maximal regionы only, which fully correspond to the representation discussed in the previous paragraph.

Let us consider another example of an event relation matrix R^L, shown in Fig. 6, constructed from an event log L over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, where $\mathcal{A}_1 = \{x_0, x_1, x_2, x_3\}$ and $\mathcal{A}_2 = \{y_0, y_1, y_2\}$.

In this event matrix, there are two p-regions $R^L(x_0 - x_1, y_0 - y_2)$ with the minimum event relation $x_1 < y_0$ and $R^L(x_0 - x_3, y_0 - y_1)$ with the minimum event relation $x_3 < y_0$. However, since $x_0 < x_1 < x_2 < x_3$, there is enough to keep $x_3 < y_0$, which will automatically satisfy $x_1 < y_0$ because x_3 occurs after x_1. This agrees with the transitivity of the precedence relation. The redundancy of these event relations can be easily shown in the corresponding agent models (see Fig. 7). We do not need to add a place between transitions x_1 and y_0 having a place between transitions x_2 and y_0.

Transition x_3 will fire only after transition x_1. Thus, adding the direct channel place between transitions x_1 and y_1 will not introduce new event relations different from those already present in the matrix from Fig. 6, unless this channel is not necessary according to the practical requirements.

	y_0	y_1	y_2
x_0	-1	-1	-1
x_1	-1	-1	-1
x_2	-1	-1	0
x_3	-1	-1	0

Fig. 6. An event relation matrix with two overlapping p-regions

The same transitivity principle can also be applied to the case of two overlapping f-regions. The example of an event relation matrix with two overlapping f-regions is shown in Fig. 8. The minimum event relation $x_0 > y_3$ will cover all event relations in both f-regions.

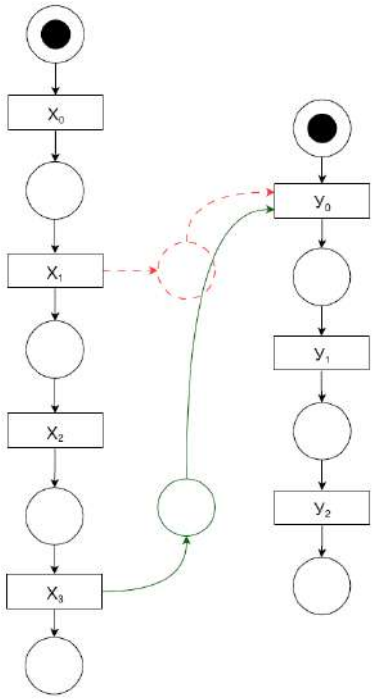


Fig. 7. Redundant channel according to the event matrix shown in Fig. 6

Note that the localization of the minimum in a region of an event relation matrix R^L actually boils down to finding the cell $r_{i,j}$, s.t.:

- if $r_{i,j} = -1$, where $r_{i+1,j} \neq -1$ and $r_{i,j-1} \neq -1$, then $r_{i,j}$ is the minimum of a p-region in R^L with the corresponding event relation $a_i < a_j$;
- if $r_{i,j} = 1$, where $r_{i-1,j} \neq 1$ and $r_{i,j+1} \neq 1$, then $r_{i,j}$ is the minimum of an f-region in RL with the corresponding event relation $a_i > a_j$.

Thus, the main scheme for the compositional discovery of a process model from an event log L over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ of a multi-agent system using minimal event relations in the event relation matrix R^L includes the following steps:

1) population of the square event relation matrix R_0^L (Algorithm 1) and selection of columns and rows (for R^L) with the actions corresponding to different agents;
2) identification of minimum event relations in p-regions and f-regions in R^L;
3) discovery of individual agent process models N_1 and N_2 from projected event logs $L_{\mathcal{A}_1}$ and $L_{\mathcal{A}_2}$, respectively;
4) introduction of channel places between transitions in N_1 and N_2 corresponding to the events associated by the minimal event relations constructed at step 2.

	y_0	y_1	y_2	y_3
x_0	0	0	1	1
x_1	0	0	1	1
x_2	1	1	1	1
x_3	1	1	1	1

Fig. 8. An event relation matrix with two overlapping f-regions

Individual agent models can be discovered from projected event logs using any existing process discovery algorithm. We recommend to use Inductive miner [10], since it can guarantee the perfect fitness of a discovered model. The soundness of the compositional discovery procedure presented above is formalized in the following Theorem 1, where we prove that a process model of a multi-agent system inherits the perfect fitness of agent models discovered from projected event logs. In other words, a process model obtained by adding channel places between transitions in the individual agent models with respect to the minimal event relations can execute all traces in the event log L of a multi-agent system.

Theorem 1. Let L be an event log of a multi-agent system over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. Let $E \subseteq (\mathcal{A}_1 \times \mathcal{A}_2) \cup (\mathcal{A}_2 \times \mathcal{A}_1)$ be the set of event pairs, which correspond to the minimum event relations extracted from the event relation matrix R^L. If N_i is a GWF-net discovered from the projection $L_{\mathcal{A}_i}$, such that it perfectly fits $L_{\mathcal{A}_i}$ with $i = 1, 2$, then N obtained from N_1 and N_2 by introducing channel places between transition pairs corresponding to event pairs in E perfectly fits L as well.

Proof. The proof is done by contradiction. Assume $N = (P, T, F, m_0, m_f)$ does not perfectly fits L. Consider a pair $(a_i, a_j) \in E$, which corresponds to the minimal event relation $a_i < a_j$. Let $\sigma \in L$ be a trace of the event log L, which contains a_i and a_j that N cannot execute. Since $a_i < a_j$, $\sigma = \sigma' a_i \sigma'' a_j \sigma'''$. Transitions $t_i, t_j \in T$ corresponding to events a_i and a_j are connected in N, such that there is a place $c \in P$, where $(t_i, c), (c, t_j) \in F$. If N cannot execute σ, then transition t_j should be able to fire before t_i, which will result in $\sigma = \sigma' a_j \sigma'' a_i \sigma'''$. This contradicts the correct configuration of the trace $\sigma = \sigma' a_i \sigma'' a_j \sigma'''$. Thus, the initial assumption that N does not perfectly L is wrong. Hence, N obtained by adding corresponding channels between transitions in N_1 and N_2 perfectly fits L.

Here, we considered the analysis of acyclic interactions between agents with sequential and conflict-free behavior. However, we can also generalize our approach to agents with conflicting (alternative) and parallel branches.

It is necessary to extend the proposed collection of event relations with the conflicting relation. Two actions a_1 and a_2 are in conflict (denoted by $a_1 \# a_2$ and 2 for the square matrix R_0^L) if for every trace in an event log a_1 and a_2 never occur together. Conflicting and parallel actions can be involved in the asynchronous interaction among agents.

Application of our approach requires separate investigation of sequential parts in agent behavior recorded in a log for the proper construction of regions in the corresponding matrix with ordered

actions. This is by analogy with the identification of sequential components in GWF-nets (recall the state machine decomposability discussed in Section 2).

For example, Fig. 9 shows the acyclic interaction between N_1 and N_2, where N_1 has the conflict between transitions x_3 and x_5. In an event log, actions x_3 and x_5 will never occur in the same trace. Using R_0^L we can identify maximal sequential parts in the behavior of N_1, i.e., $x_2 < x_3$ and $x_4 < x_5$, and construct two inter-agent matrices to localize minimal event relations in corresponding regions. Two minimal event relations $y_2 < x_3$ and $y_2 < x_5$ with the common event y_2 are ensured with a single channel place a connecting transitions w.r.t. the relation direction.

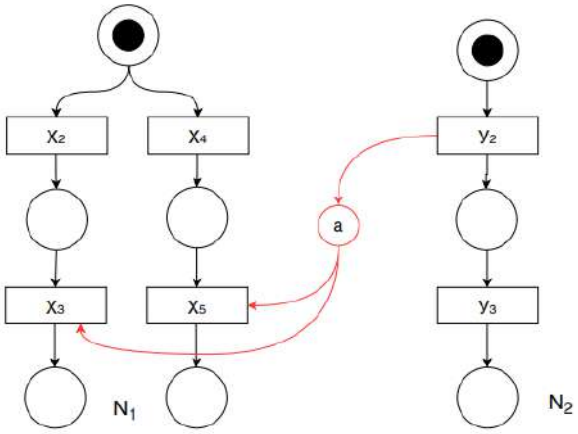


Fig. 9. Acyclic interaction with choice in the agent behavior

Using a similar reasoning, we can analyze asynchronous interactions involving different parallel branches in the behavior of agents. In this case, the minimal relations with the common events are modeled by individual channel places, since, for parallel actions, the occurrence of one does not exclude the occurrence of the others.

In the following section, we consider asynchronous interactions among agents, s.t. actions used for the message exchange are involved in a cycle. The direct analysis of causality relation is not enough for cyclic behavior, since events within a cycle can be recorded in an event log in any order.

4. Localizing Cyclic Agent Interactions

In this section, we consider the problem of identifying the pairs of events in an event log of a multi-agent system involved intro the cyclic interaction between different agents. Cyclic interaction implies that the actions corresponding to the asynchronous message exchange are executed within a cycle in the agent behavior. We cannot directly use the minimal causality relations proposed in the previous section, since actions within cycles in different agents will be recorded in an event log in any order.

4.1 Bounded Asynchronous Channels

The cyclic interaction is directly connected with the problem of the boundedness in Petri net theory. Consider an example of cyclic interaction shown in Fig. 10. The cycle in N_1 sends messages to the cycle in N_2 via the single channel a.

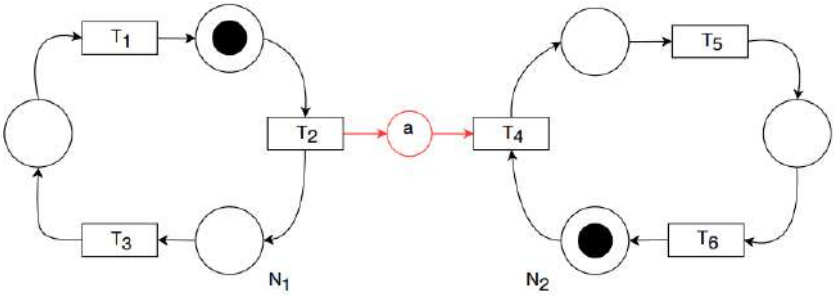


Fig. 10. An unbounded asynchronous channel

The problem with this channel place a is that N_1 can put messages to place a infinitely many times, which will lead to the possibility of the unbounded number of messages in a. As a result, the complete system will have infinitely many different reachable states.

To avoid the problem of the unboundedness, we can introduce an additional place into the model of a multi-agent system with two interacting agents. This place will act as a "limiter" of the number of messages an asynchronous channel can store.

For example, if we add place b, as shown in Fig. 11, the maximum number of messages that can be put to place a by N_1 will not exceed 1. Such places are called complement in Petri nets, since they mirror the direction of arcs connected with the channel place.

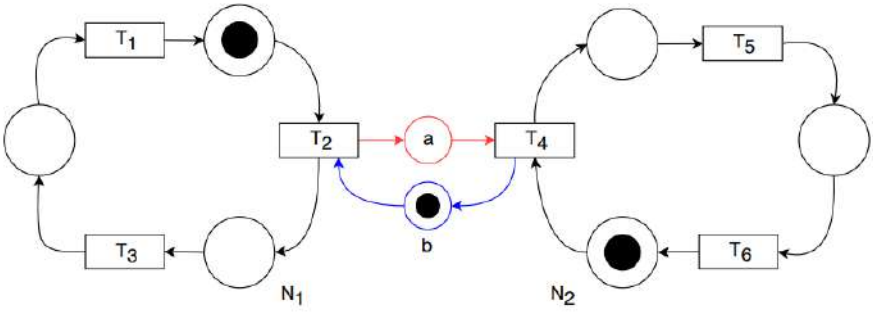


Fig. 11. An asynchronous channel with the bound

In fact, the number of tokens in the complement place we add to bound an asynchronous channel correspond to the maximum number of messages this asynchronous channel can store. In the following paragraph, we show our approach to the analysis of cyclic interactions between agents in a multi-agent system with respect to the maximum number of messages a candidate asynchronous channel place can store.

4.2. Algorithm for Localizing Cyclic Asynchronous Interactions and Channel Bounds

In the case of the cyclic asynchronous interactions, we cannot directly refer to the minimum event relations, since all involved actions can potentially be recorded in any order in an event log. For example, by simulating the net from Fig. 11, we can obtain $t_2 < t_4$ as well as $t_4 < t_2$. Instead, we are going to consider the number of occurrences of events in an event log to devise the maximum number of messages an asynchronous channel can handle.

For what follows, let L be an event log of a multi-agent system with two asynchronously interacting agents over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. We isolate only the cyclic behavior of agents in these sets $\mathcal{A}_1$ and $\mathcal{A}_2$, since the acyclic part can be analyzed before using the algorithm described in Section 3. To avoid

the ambiguity, we assume additionally that actions $\mathcal{A}_1$ represent the behavior of an agent sending messages, while the actions $\mathcal{A}_2$ — the behavior of an agent receiving messages.

The main idea of our approach is to analyze pairs of actions in $\mathcal{A}_1 \times \mathcal{A}_2$ to count the maximum number of messages. If in a trace of L the occurrence of an event $a_1 \in \mathcal{A}_1$ is recorded, then the bound in the number of messages decreases by 1. If in a trace of L the occurrence of an event $a_2 \in \mathcal{A}_2$ is recorded, then the bound in the number of messages increases by 1.

We assume that an asynchronous channels stores $k \geq 0$ messages initially. Algorithm 2 shows how to analyze the pairs of events in $\mathcal{A}_1 \times \mathcal{A}_2$ according to their behavior with respect to increasing and decreasing k. This algorithm produces the range, i.e., the minimum and maximum number of messages an asynchronous channel between a concrete pair of events can process.

Consider the example of using Algorithm 2 for the event log of a multi-agent system L (see Table 2) over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, where $\mathcal{A}_1 = \{t_4, t_5, t_6\}$ and $A_2 = \{t_1, t_2, t_3\}$.

Table 2. An event log of a multi-agent system with four traces

Trace 1	$t_4t_5t_6t_4t_5t_2t_3t_6t_4t_5t_6t_4t_5t_2t_3t_6t_1t_2t_3t_1t_2t_3t_4t_1t_5t_6t_2t_3t_1t_2t_3t_1t_2t_3t_1t_4t_2t_3t_5$ $t_1t_6t_4t_5t_6t_4t_5t_6t_4t_2t_5t_3t_6t_1$
Trace 2	$t_2t_3t_1t_2t_3t_1t_2t_3t_4t_1t_5t_2t_6t_3t_4t_1t_5t_6t_4t_2t_5t_3t_1t_6t_2t_4$
Trace 3	$t_2t_3t_1t_2t_3t_1t_2t_3t_1t_4t_1t_5t_6t_4t_1t_5t_6t_4t_1t_5t_6t_4t_1t_5t_6t_4t_1t_5t_6t_4t_1t_5t_6$
Trace 4	$t_4t_1t_5t_6t_4t_1t_5t_6t_2t_3t_1t_2t_3t_1t_2t_3t_1t_2t_3t_1t_2t_3t_1$

The result of computing the minimum and maximum number of messages for different event pairs in Trace 1 in this event log is presented in Table 3.

For instance, we consider the pair of events (t_4, t_1) of transitions between which we aim to add a bounded asynchronous channel place. We check the minimum and maximum number of messages for all traces in the event log from Table 2, as shown in Table 4.

Table 3. Applying Algorithm 2 to Trace 1 in the log from Table 2

Event pair	Minimum	Maximum
(t_1, t_4)	$k-3$	$k+2$
(t_1, t_5)	$k-4$	$k+1$
(t_1, t_6)	$k-4$	$k+2$
(t_2, t_4)	$k-2$	$k+3$
(t_2, t_5)	$k-3$	$k+3$
(t_2, t_6)	$k-2$	$k+3$
(t_3, t_4)	$k-3$	$k+2$
(t_3, t_5)	$k-3$	$k+2$
(t_3, t_6)	$k-3$	$k+2$

To cover the complete event log from Table 2, we need to construct the range for the channel between events t_4 and t_1 uniting the individual ranges for all traces. Thus, according to Table 4, the range of the number of messages that can be handled by the asynchronous channel between transitions t_4 and t_1 is $[k-3; k+3]$. The length of this range is $k+3-(k-3)=6$. Therefore, the maximum number of messages that can be stored in the channel between t_4 and t_1 is bounded by 6.

Note also that, since the left border of this range $k-3$, initially the channel place between t_4 and t_1 should have 3 tokens in it, because the number of tokens in places of a Petri net cannot go below 0. This is also caused by the fact that in Trace 2 of the event log from Table 2 the agent receiving messages operates before the one who sends messages.

Algorithm 2: Analyzing cyclic interactions in a trace

Input: $\sigma \in L$ – a trace in an event log over $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$, where $\mathcal{A}_1 \cup \mathcal{A}_2 = \varnothing$

Output: Minimum $min(p)$ and maximum $max(p)$ number of messages for every $p = (a_1, a_2) \in \mathcal{A}_1 \times \mathcal{A}_2$ a channel between a_1 and a_2 may process

```
foreach a1 ∈ A1 do
    foreach a2 ∈ A2 do
        maxK ← k, minK ← k, current ← k
        foreach e_i ∈ σ = e1 e2 ... en do
            if e_i = a1 then
                current ← k − 1
            end
            if e_i = a2 then
                current ← k + 1
            end
            maxK ← MAX(current, maxK)
            minK ← MIN(current, minK)
        end
    end
    min(a1, a2) ← minK, max(a1, a2) ← maxK
end
```

We have everything to construct the model of a multi-agent system with two agents exchanging messages through actions t4 and t1 within cyclic sequential behavior regarding the event log from Table 2. Fig. 12 shows the corresponding process model for this multi-agent system, where N_1 is the agent sending messages with transitions t_4, t_5, t_6, and N_2 — receiving messages with transitions t_1, t_2, t_3.

Table 4. The number of messages in the channel connecting t_4 and t_1

	Minimum	**Maximum**
Trace 1	$k-3$	$k+2$
Trace 2	k	$k+2$
Trace 3	$k-2$	$k+3$
Trace 4	$k-2$	$k+3$

We note that the similar analysis can be done for any pair of transitions representing the behavior of sending and receiving agents, s.t. one can add an asynchronous channel between them in different ways, unless there is an additional information on actions provided. For instance, one can choose those transitions with the channel the capacity of which does not exceed 1 (for safe Petri nets). In addition, as in the case of the acyclic interaction, it is possible to analyze the cyclic behavior of agents with parallel and alternative behavioral constructs inside cycles by checking interactions between separate sequential components.

Moreover, the same property on preserving the perfect fitness of the individual agent models (see Theorem 1) will also hold for the cyclic interaction, since we add channel places between transitions in the strict accordance with an initial event log.

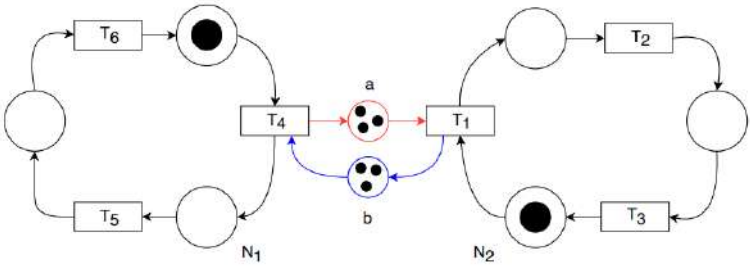


Fig. 12. A multi-agent system with two interacting agents with cyclic behavior

5. Experimental Evaluation

This section reports the key outcomes obtained from the series of experiments conducted to evaluate the proposed approach to the identification of the pairs of events involved into the acyclic and cyclic interactions among different agents in a multi-agent system.

5.1 Layout of Experiments

We compared process models discovered by our approach and directly from an event log of a multi-agent system. We also considered a specific case of a process model with "disconnected" agents, i.e., we do not add asynchronous channels between them.

Within the experimental evaluation, we used the synthetic event logs of multi-agent systems recording different ways of agent asynchronous interactions provided in [11]. They were also used to test the compositional approach to discovering architecture-aware process model of multi-agent systems [6]. This dataset was constructed with respect to various widespread service interaction patterns described in [12].

Thus, process models of multi-agent systems obtained by our approach to introducing channels were compared with the following other models:

1) *reference* models, also provided in [11], which represent the ideal model of a multi-agent system with the minimum number of asynchronous channels;
2) *disconnected agent* models, where individual agent models discovered from projected event logs are put together without adding any asynchronous channels;
3) monolithic models discovered from directly event logs.

We characterized these models according to the following two quality dimensions:

1) *precision* evaluating the extra amount of behavior allowed by a process models regarding the behavior recorded in an event log (see the gray area in Fig. 13);
2) the number of asynchronous channels connecting transitions in the models of different agents.

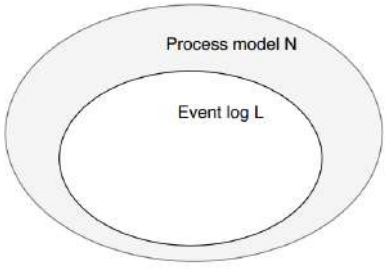


Fig. 13. The behavior of a process model and traces in an event log

The perfect fitness of discovered process models is guaranteed by our approach and by the paper [6]. A model with the disconnected agent behavior also ensures the perfect fitness, since the concurrent execution of fully independent agents can also cover all possible ways of their asynchronous interactions. Therefore, we did not need to measure the fitness of considered process models. As for the precision, we used the approach from [13] as the one, which provides the balanced estimation of this quality dimension. The experimental evaluation was supported by the ProM software [14].

5.2 Experiment Results and Discussion

Table 5 reports the results on comparing the quality of process models discovered from an event log of a multi-agent system using our approach with the quality of directly discovered models (monolithic) and models with the disconnected agent behavior. The dataset [11] used in our experiments contains seven different event logs of multi-agent system corresponding to different ways of acyclic (IP-1, ..., IP-6) and cyclic (IP-7) patterns of asynchronous interactions. We also did not evaluate the number of channels in monolithic process models of multi-agent systems, since in the structure of such a model one cannot unambiguously identify the behavior of individual agents and asynchronous channel places.

Table 5. Experimental results: the number of asynchronous channels and precision evaluation

Interaction		**Reference**		**Disconnected**	**Monolithic**	**Our approach**	
		Channels	**Precision**	**Precision**	**Precision**	**Channels**	**Precision**
Acyclic	**IP-1**	1	0.7156	0.6949	0.5825	14	0.8109
	IP-2	2	0.4014	0.3719	0.3880	33	0.5337
	IP-3	2	0.7545	0.7097	0.8984	26	0.8861
	IP-4	2	0.7589	0.6752	0.6684	10	0.8420
	IP-5	4	0.3902	0.3503	0.1342	39	0.5724
	IP-6	4	0.5636	0.5256	0.6849	34	0.7034
Cyclic	**IP-7**	3	0.8165	0.5945	0.1327	5	0.6782

According to the experimental results provided in Table 5, we may conclude the following. Firstly, our approach detects considerably more “points” of the asynchronous interactions between different agents compared to the ideal reference model. A finite sequential record of the concurrent execution of relatively independent agents cannot cover all possible scenarios. Thus, there are more candidate relations among event pairs that can be considered for adding asynchronous channel places between the corresponding transitions. We can further analyze all found minimum event relations from the point of view on their frequencies w.r.t. an initial event log to exclude some of them. Secondly, process models obtained by our approach exhibits the increase in the precision estimations, since introduction of other asynchronous channels decreases the amount of extra behavior allowed by a model and not recorded in a log. Thirdly, we generally outperform the quality of the monolithic process model the structure of do not correspond to the architecture of a multi-agent system regarding the individual agent behavior and their interactions.

We believe that increasing the number of traces in an event log will bring the quality of process models obtained by adding channels using our approach closer to the evaluations of reference models, since an event log will exhibit more different execution scenarios. As one of the possible directions of future research, we will consider the analysis of connections between the precision of agent models and of system models obtained by our approach based on event relation.

6. Related Work

As we mentioned in Introduction, different algorithms were proposed for the computer-aided discovery of process models from event logs. The most popular ones include Inductive miner [10],

Fuzzy miner [15], Region Theory-based miner [16], and Genetic miner [17]. These algorithms can guarantee that discovered process models will exhibit certain properties. For example, Inductive miner can guarantee perfect fitness and soundness of discovered workflow nets. In the recent study [7], the authors gave an extensive review and comparison of process discovery algorithms. Note that these algorithms are aimed to tackle different internal limitations of event data representation rather than to analyze interactions among different information system components.

The quality of discovered process models takes an important part in choosing an algorithm for discovering process models from event logs. Conformance checking [7] provides several dimensions that allow one to evaluate the correspondence between a model and an event log (fitness, precision, generalization), and the structure of a discovered model (simplicity). Researchers stress that there is a lack of universally applicable properties and requirements that can constitute the formal basis for computing conformance checking dimensions [7, 18]. Thus, our study also considers the formal analysis of preserving the perfect fitness of agent models discovered from filtered logs in a multi-agent system models with introduced asynchronous channels recovered using event relations.

The problem of discovering process models with a clear structure is studied from different perspectives. Inductive miner produces well-structured process models that are recursively constructed from “building blocks” representing standard behavioral constructs: sequential, cyclic, parallel, and alternative execution of actions. A series of papers [19, 20, 21] proposed different approaches to improving the structure of discovered models by the additional localization of the environment of events in a log and by composing fragments of regular and frequent behavior with the rare “exceptional” scenarios. Discovery of hierarchical process models, where a high-level event represents a sub-process, was studied in [4]. The identification of low-level and high-level events in an event log is a natural way to improve the structural representation of a process model. The paper [3] proposed a novel approach to discover object-centric Petri nets from event logs. Interactions of objects is represented through complex synchronizations which allow one to model consumption and production of objects of different types. Compositional discovery of behaviorally correct and “architecture-aware” process models from event logs of multi-agent systems was studied in [6]. Using interface patterns and structural property-preserving mapping helped to achieve the clear structure of a model reflecting independent behavior of agents and their communication.

Our study continues [6] in a way that we are trying to analyze and identify “points” of asynchronous interactions — actions involved in the asynchronous message passing between agents — directly from event logs. Based on the causality relations among events in a log, we can find, for example, pairs of actions that are always executed in a fixed order. Such actions are then considered to be the candidates to represent send-receive operations within the asynchronous interaction. Then we may relax the requirement on the manual selection of interface patterns, as originally proposed.

Patterns are typically used in the software development as the collection of best practices and recurring development scenarios [22]. Frequently used control-flow constructs in business process modeling — workflow patterns — were systematically studied in [23]. In [12, 24], the authors generalized workflow patterns for modeling widespread correct service interactions in complex and large-scale systems. Within the context of process discovery, several papers also proposed different approaches for the analysis of behavioral patterns in event logs, including, among the others, [25, 26], but these patterns were not considered from the point of view of interactions among different information system components.

7. Conclusion

In this paper, we considered the problem of discovering a process model in terms of a generalized workflow net from an event log of a multi-agent system with the understandable structure reflecting the architecture of a system. A model of a multi-agent system is obtained from a composition of individual agent models through the introduction of asynchronous channels. To identify transitions in agent models to be connected via a channel place, we analyze causal relations between events

recorded in an event log. Within the asynchronous agent interactions, several actions of one agents are executed before certain actions of the other. This idea helped us to localize the so-called minimum event relations corresponding to the occurrence of actions executed by different agents. The pairs of events representing these minimum relations can be seen as "points" of the asynchronous communication between agents. Transitions corresponding to these events can be connected with an asynchronous channel place. We also showed that certain minimum event relations can cover other minimum relations between events in a log.

The pair-wise analysis of relations between events recorded in an event log was based on matrices with rows and columns representing events. Matrix representation of event logs was used in process mining in different contexts (cf. the footprint matrix in the basic α-algorithm [27] and the analysis of unchanged sections in BPMN models [28]).

We separately considered the cases of the acyclic and cyclic asynchronous interactions, since, within the latter one, events can be recorded in any possible order. To localize events in the cyclic communication, we analyzed the number of event occurrences regarding the maximum number of messages that a potential asynchronous channel can handle. This allows us to achieve the boundedness, i.e., the finite number of reachable states, in a complete process model of a multi-agent system.

The correctness of the proposed approach to adding asynchronous channels between behavioral models of individual agents is justified by the fact that we preserve the perfect fitness, i.e., the ability to execute all traces in the event log of a multi-agent system, of agent model in a complete system model. We conducted a series of experiments to evaluate our approach. The experimental results demonstrate the overall improvement in process models discovered by adding asynchronous channels in comparison to models directly discovered from event logs of multi-agent systems.

As for the future research, we plan to continue it in the following directions. Firstly, we would like to consider more complex ways of the asynchronous communications, including, for instance, message broadcasting. Secondly, we also intend to make a deeper analysis of the preservation of behavioral properties, including deadlock-freeness, in a process model of a multi-agent system obtained from individual agent models connected by asynchronous channel places. For example, we need to avoid the introduction of channels leading to the "circular wait", as shown in Fig. 14, where N_1 waits for N_2, while N_2 waits for N_1 at the same time. Finally, we plan to conduct more experiments using real-life event logs.

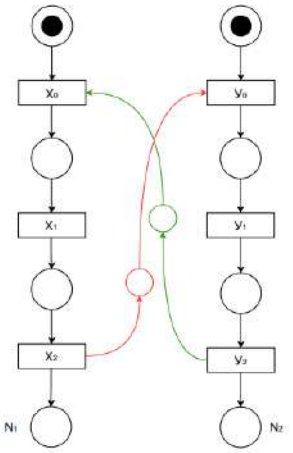


Fig. 14. Asynchronous interaction may result in a deadlock

References

[1] W. van der Aalst. Process Mining: Data Science in Action. Springer, Heidelberg, 2016. DOI: 10.1007/978-3-662-49851-4.

[2] W. Reisig. Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies. Springer, Heidelberg, 2013. DOI: 10.1007/978-3-642-33278-4.

[3] W. van der Aalst and A. Berti. Discovering Object-Centric Petri Nets. Fundamenta Informaticae, vol. 175, pp. 1–40, 2020. DOI: 10.3233/FI-2020-1946.

[4] A. Begicheva and I. Lomazova. Discovering High-Level Process Models from Event Logs. Modeling and Analysis of Information Systems, vol. 24, no. 2, pp. 125–140, 2017. DOI: 10.18255/1818-1015-2017-2-125-140.

[5] C. Li, S. van Zelst, and W. van der Aalst. An Activity Instance Based Hierarchical Framework for Event Abstraction. In 2021 3rd International Conference on Process Mining (ICPM), 2021, pp. 160–167. DOI: 10.1109/ICPM53251.2021.9576868.

[6] R. Nesterov, L. Bernardinello, I. Lomazova, and L. Pomello. Discovering architecture-aware and sound process models of multi-agent systems: a compositional approach. Software & Systems Modeling, vol. 22, pp. 351–375, 2023. DOI: 10.1007/s10270-022-01008-x.

[7] A. Augusto, R. Conforti, M. Dumas, M. Rosa, F. Maggi, A. Marrella, M. Mecella, and A. Soo. Automated Discovery of Process Models from Event Logs: Review and Benchmark. IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 4, pp. 686–705, 2019. DOI: 10.1109/TKDE.2018.2841877.

[8] J. Carmona, B. van Dongen, A. Solti, and M. Weidlich, Conformance Checking: Relating Processes and Models. Springer, Cham, 2018. DOI: 10.1007/978-3-319-99414-7.

[9] W. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In Business Process Management: Models, Techniques, and Empirical Studies. Lecture Notes in Computer Science, vol. 1806. Springer, Heidelberg, 2000, pp. 161–183. DOI: 10.1007/3-540-45594-9_11.

[10] S. Leemans, D. Fahland, and W. van der Aalst. Discovering Block-Structured Process Models from Event Logs – A Constructive Approach. In Application and Theory of Petri Nets and Concurrency (PETRI NETS 2013). Lecture Notes in Computer Science, vol. 7927. Springer, Heidelberg, 2013, pp. 311–329. DOI: 10.1007/978-3-642-38697-8_17.

[11] R. Nesterov, “Compositional discovery of architecture-aware and sound process models of multi-agent systems: experimental: data experimental data. (version 1) [data set].” [Online]. Available: https://doi.org/10.5281/zenodo.5830863.

[12] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns. In Business Process Management (BPM 2005). Lecture Notes in Computer Science, vol. 3649. Springer, Heidelberg, 2005, pp. 302–318. DOI: 10.1007/11538394_20.

[13] J. Munoz-Gama and J. Carmona. A Fresh Look at Precision in Process Conformance. In Business Process Management (BPM 2010). Lecture Notes in Computer Science, vol. 6336. Springer Heidelberg, 2010, pp. 211–226. DOI: 10.1007/978-3-642-15618-2_16.

[14] “ProM Tools,” [Online]. Available: https://www.promtools.org/doku.php.

[15] C. Gunther and W. van der Aalst. Fuzzy Mining – Adaptive Process Simplification Based on Multi-Perspective Metrics. In Business Process Management (BPM 2007). Lecture Notes in Computer Science, vol. 4714. Springer, Heidelberg, 2007, pp. 328–343. DOI: 10.1007/978-3-540-75183-0_24.

[16] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In Business Process Management (BPM 2007). Lecture Notes in Computer Science, vol. 4714. Springer, Heidelberg, 2007, pp. 375–383. DOI: 10.1007/978-3-540-75183-0_27.

[17] W. van der Aalst, A. de Medeiros, and A. Weijters. Genetic Process Mining. In Applications and Theory of Petri Nets (ICATPN 2005). Lecture Notes in Computer Science, vol. 3536. Springer, Heidelberg, 2005, pp. 48–69. DOI: 10.1007/11494744_5.

[18] W. van der Aalst. Relating Process Models and Event Logs – 21 Conformance Propositions. In Proceedings of the International Workshop ATAED-2018. CEUR Workshop Proceedings, vol. 2115. CEURWS.org, 2018, pp. 56–74.

[19] A. Kalenkova, I. Lomazova, and W. van der Aalst. Process model discovery: A method based on transition system decomposition. In Application and Theory of Petri Nets and Concurrency (PETRI

NETS 2014). Lecture Notes in Computer Science, vol. 8489. Springer, Cham, 2014, pp. 71–90. DOI: 10.1007/978-3-319-07734-5_5.

[20] A. Kalenkova and I. Lomazova. Discovery Of Cancellation Regions Within Process Mining Techniques. Fundamenta Informaticae, vol. 133, pp. 197–209, 2014. DOI: 10.3233/FI-2014-1071.

[21] W. van der Aalst, A. Kalenkova, V. Rubin, and E. Verbeek. Process Discovery Using Localized Events. In Application and Theory of Petri Nets and Concurrency. Lecture Notes in Computer Science, vol. 9115. Springer, Cham, 2015, pp. 287–308. DOI: 10.1007/978-3-319-19488-2_15.

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

[23] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. Distributed and Parallel Databases, vol. 14, pp. 5–51, 2003. DOI: 10.1023/A:1022883727209.

[24] D. Campagna, C. Kavka, and L. Onesti. BPMN 2.0 And The Service Interaction Patterns: Can We Support Them All? In Software Technologies (ICSOFT 2014). Communications in Computer and Information Science, vol. 555. Springer, Cham, 2015, pp. 3–20. DOI: 10.1007/978-3-319-25579-8_1.

[25] S. Suriadi, R. Andrews, A. ter Hofstede, and M. Wynn. Event logs imperfection patterns for process mining: towards a systematic approach to cleaning event logs. Information Systems, vol. 34, pp. 132–150, 2017. DOI: 10.1016/j.is.2016.07.011.

[26] M. Acheli, D. Grigori, and M. Weidlich. Discovering and Analyzing Contextual Behavioral Patterns from Event Logs. IEEE Transactions on Knowledge and Data Engineering, vol. 34, no. 12, pp. 5708–5721, 2022. DOI: 10.1109/TKDE.2021.3077653.

[27] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering, vol. 16, no. 9, pp. 1128–1142, 2004. DOI: 10.1109/TKDE.2004.47.

[28] K. Artamonov and I. Lomazova. What Has Remained Unchanged in Your Business Process Model? In 2019 IEEE 21st Conference on Business Informatics (CBI), 2019, pp. 551–558. DOI: 10.1109/CBI.2019.00070.

Информация об авторах / Information about authors

Анастасия Андреевна ШЕРСТЮГИНА – студентка бакалавриата факультета компьютерных наук НИУ Высшая Школа Экономики (ВШЭ), стажер-исследователь научно-учебной лаборатории процессно-ориентированных информационных систем (ПОИС) НИУ ВШЭ. Область научных интересов: моделирование и формальный анализ поведения процессов в информационных системах с помощью сетей Петри и других формализмов, объектно-ориентированное программирование и архитектура информационных систем.

Anastasiya SHERSTYUGINA is a bachelor student at the faculty of computer science in HSE University and a research assistant at the Laboratory for Process-Aware Information Systems (PAIS Lab), HSE University. Her research interests mainly include modeling and analysis of process behavior in information systems using Petri nets and other related formalisms, object-oriented programming and architecture of information systems.

Роман Александрович НЕСТЕРОВ – старший преподаватель факультета компьютерных наук НИУ ВШЭ, младший научный сотрудник научно-учебной лаборатории ПОИС НИУ ВШЭ. Имеет степень кандидата компьютерных наук (физико-математические науки) НИУ ВШЭ (2022 г.). Область научных интересов: теория параллелизма, сети Петри, теория категорий, формальные методы моделирования и верификации сложно организованных информационных систем.

Roman NESTEROV is a senior lecturer at the faculty of computer science in HSE University and a junior researcher at the PAIS Lab, HSE University. He holds a PhD degree in Computer Science awarded by HSE University in 2022. His research interests include concurrency and category theory, Petri nets, formal methods for modeling and verifying complex information systems.

DOI: 10.15514/ISPRAS-2023-35(3)-2

Writable PSI Generator for a Multi-Language IDE Platform

[1] *A.S. Bozhnyuk*, ORCID: 0009-0003-4826-6609 <*bozhnyuks@mail.ru*>
[2] *A.A. Zakharov*, ORCID: 0009-0005-0087-0633 <*lynxsm@gmail.com*>
[1] *H.V. Tropin*, ORCID: 0009-0006-2910-3961 <*niktrop@yandex.ru*>
[1] *M.V. Volkov*, ORCID: 0000-0001-8672-7750 <*mvvolkov@mail.ru*>

[1] *St. Petersburg State University,*
7/9, University Embankment, Saint Petersburg, 199034, Russia.
[2] *Tula State University,*
92, Lenin's avenue, Tula, 300012, Russia

Abstract. Any state-of-the-art integrated development environment (IDE) should provide software developers with services for quick and correct code transformation. Such services can be used both for program refactoring to improve its quality and for quick fixing of syntax errors in source code. This paper presents the method of constructing a subsystem that makes it possible to create the services described above and also has the property of fast extensibility to support different programming languages. The method of transformation of Program Structure Interface (PSI) - a special data structure, which provides API for development of IDE-services, is proposed. Besides, a method of generating types for PSI in accordance with the syntax of the supported programming language is proposed. The approach is developed for a multi-language platform of a large telecommunications company. Refactoring and Quick Fix features are implemented using on the proposed generator for two IDEs: a Python IDE and a Java IDE.

Keywords: Integrated Development Environment (IDE); Development Services; Program Structure Interface (PSI); Application Program Interface (API); Refactoring, Quick Fix.

For citation: Bozhnyuk A.S., Zakharov A.A., Tropin N.V., Volkov M.V. Writable PSI Generator for a Multi-Language IDE Platform. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 33-46. DOI: 10.15514/ISPRAS-2023-35(3)-2

Генератор дерева PSI с возможностью записи для мультиязыковой платформы IDE

[1] *А.С. Божнюк*, ORCID: 0009-0003-4826-6609 <*bozhnyuks@mail.ru*>
[2] *А.А. Захаров*, ORCID: 0009-0005-0087-0633 <*lynxsm@gmail.com*>
[1] *Н.В. Тропин*, ORCID: 0009-0006-2910-3961 <*niktrop@yandex.ru*>
[1] *М.В. Волков*, ORCID: 0000-0001-8672-7750 <*mvvolkov@mail.ru*>

[1] *Санкт-Петербургский государственный университет,*
199034, Россия, Санкт-Петербург, Университетская наб., д. 7-9.
[2] *Тульский государственный университет,*
300012, Россия, Тула, пр. Ленина, 92

Аннотация. Любая современная интегрированная среда разработки (IDE) должна предоставлять разработчикам программного обеспечения сервисы для трансформации кода. Такие сервисы могут использоваться как для рефакторинга программы с целью улучшения её качества, так и для быстрого

исправления синтаксических ошибок в исходном коде. Данная работа предлагает технологию разработки подсистемы, которая позволяет создавать такие сервисы для различных IDEs и языков программирования. Предложен способ трансформации дерева Program Structure Interface (PSI) – специальной структуры данных, предоставляющей API для разработки IDE-сервисов. Помимо этого, предложен способ генерации типов для PSI сообразно синтаксису поддерживаемого языка программирования. Подход разработан для мультиязыковой платформы, создаваемой в рамках крупной технологической компании. На его основе были разработаны сервисы по модификации исходного кода для двух сред разработки – Java IDE и Python IDE.

Ключевые слова: Интегрированная среда разработки; IDE; средства разработки; PSI; программный интерфейс; рефакторинг; исправление кода.

Для цитирования: Божнюк А.С., Захаров А.А., Тропин Н.В., Волков М.В. Генератор дерева PSI с возможностью записи для мультиязыковой платформы IDE. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 33–46 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–2

1. Introduction

An Integrated Development Environment (IDE) is an essential tool for any programmer. Some of the most well-known and widely used IDEs are JetBrains IntelliJ IDEA and Microsoft Visual Studio, which offer a large number of features to develop high-quality software.

One of the most important tasks of the IDE is to provide developers the ability to quickly and correctly modify the source code. To achieve this, IDEs offer such features as refactoring and quick fixes. Refactoring makes it possible to restructure code while preserving its semantics, for example, to rename a class, method, and attribute, extract selected code into a method, and so on. Quick fixes, at the request of the developer, eliminate a drawback of a code fragment. An example of a quick fix is if statement simplification.

These features work with the structure of the program by analyzing and reorganizing it. The conventional way of representing a program internally is an Abstract Syntax Tree (AST) that is generated via parsing [1]. However, IDEs often need to work with additional semantic information (for example, to determine the declaration of a method or attribute by its occurrence), which would also be convenient to store in the tree. Therefore, the IDE builds another tree on top of the AST, which gives external clients (IDE features) access to such information about the program. In IntelliJ IDEA, such a tree is called Program Structure Interface (PSI) [2], and this name is used in this paper. Thus, PSI stores additional information and provides clients with a rich API, and AST is an implementation detail.

For convenience, in PSI each of its nodes has its own type according to the syntax structure of the language that this node represents, within the syntax construct type system of the programming language in which this tree is created. For example, in the context of the Java language, each node is defined by its own class (PsiFunction, PsiClass, etc.). At the same time, different IDEs implement various approaches to building such a data structure and methods of interacting with it [3, 4].

IDE features, after manipulating PSI, transfer the changes to the source code so that they become visible to the developer. To do this, text changes are generated based on the changes in the tree, which are then applied to the code.

As mentioned above, PSI is typed. This is convenient, but the types in such a tree must be accessed somehow in the source code. If the IDE, for example, is developed in Java, then one will need to create a large number of interfaces and classes for that purpose. This process is very time-consuming due to the large number of types, and therefore highly error-prone. For this reason, it is desirable to use generation, which is based on a pre-created specification of syntax construct types of the programming language.

A large telecommunications company is developing a multi-language platform for effectively creating IDEs for different programming languages. Two IDEs (for Java and Python) are being created at the moment.

The platform requires a unified system for managing the source code structure. Each specific IDE requires its own PSI tree and tools for manipulating it, as well as a system for displaying changes in the code. However, the principles of generating PSI access interfaces based on programming language construct types are universal and can be implemented within the platform and used in various IDEs.

The main contributions of the paper are as follows.

- Design of the Writable PSI Generator architecture: a mechanism for generating classes and interfaces for accessing the PSI tree, as well as a single mechanism for distributing text changes.
- Implementation of the Writable PSI Generator including a component for generation of the necessary Java interfaces and classes for the PSI tree modification system based on JSON specification, and a component for modifying the tree consisting of a persistent tree and a Rewriter, and a mechanism for obtaining text changes (the GumTree algorithm) [5].
- The Writable PSI Generator was successfully tested in the Java and Python IDEs in the implementation of a number of refactoring services and quick fixes.

The remainder of this paper is organized as follows: in Section 2, we present functional requirements and the architecture of the Writable PSI Generator. Section 3 presents system implementation details. Further, we discuss the convenience of the Writable PSI Generator and show the success of reuse in Section 4. Finally, Section 5 presents related work.

2. Architecture

The functional requirements of the Writable PSI Generator are the following.

- The system should allow for generating Java interfaces and classes for working with PSI (the main language within the multi-language platform used for developing various IDEs is Java).
- The system should allow for modifying the tree for the needs of refactorings and quick fixes.
- It should be possible to get text changes for the source code document.
- It is necessary to ensure that the system can be reused for various IDEs developed within a multi-language platform.

The Writable PSI Generator consists of two subsystems: the subsystem for transforming the PSI and obtaining text changes, and the subsystem for generating interfaces and classes access to the PSI tree.

Fig. 1 shows an UML component diagram describing the subsystem for transforming the PSI and obtaining text changes in the Writable PSI Generator.

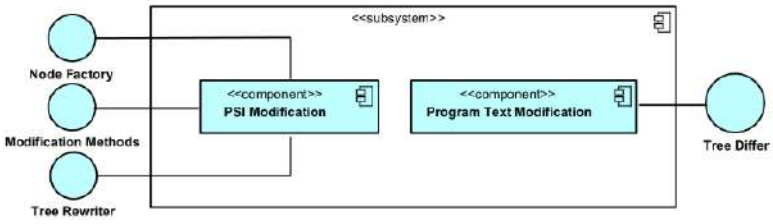


Fig. 1. Subsystem for transforming the PSI and obtaining text changes

The PSI Modification component provides external clients with different ways to modify and build new PSI nodes:

- Node Factory is a factory that provides methods for both constructing PSI nodes from other nodes and creating them from a string. This factory is generated, but at the same time it is possible for it to "manually" add additional methods.
- Modification Methods are generated methods that each interface and class contain for modifying the attributes of the syntax construct. These methods allow the client to create a new version of the node, replacing existing children.
- Tree Rewriter is an entity that allows the client to create a new copy of PSI by replacing or removing some nodes. Implements the Builder design pattern.

The Program Text Modification component provides external services with the ability to receive text changes to a document after PSI transformations. The client is provided with a Tree Differ, which, after receiving two trees, finds differences between them and creates the sequence of text changes that the client can apply to the source code document.

Fig. 2 shows a UML sequence diagram which describes the main scenario of using the subsystem for transforming the PSI and obtaining text changes.

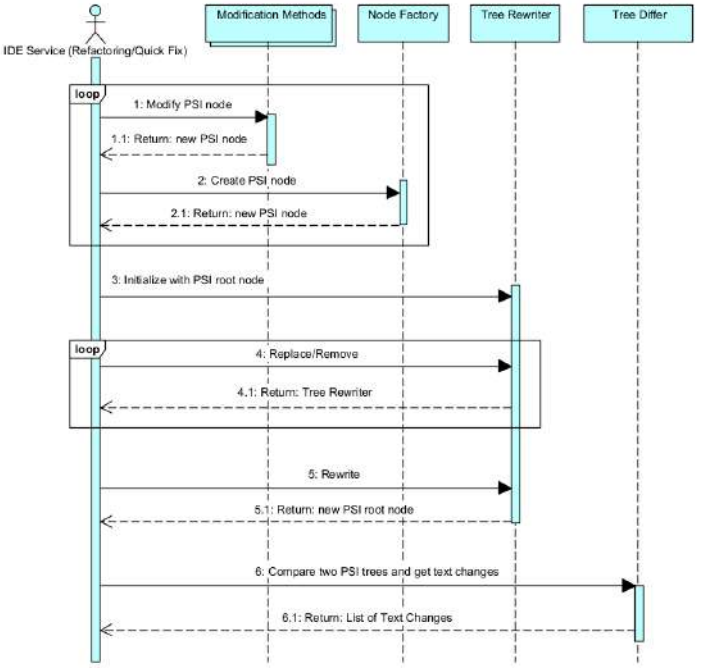


Fig. 2. The main scenario of using the subsystem for transforming the PSI and obtaining text changes

It includes the following steps:

- Modification of tree nodes (1) or construction of new nodes using the factory (2). As a result, new tree nodes are available for the external service.
- Modification of the entire PSI is performed using the Tree Rewriter interface, as a result of which a new, modified copy of the PSI is created. Firstly, the feature initializes this interface with the root of the new PSI (3). Secondly, then through the replace/remove methods it indicates which transformations need to be performed (4). Finally, using the Rewrite method (5), the modification process is activated, and as a result, the feature receives a new PSI.
- Obtaining a sequence of text changes using the Tree Differ interface, which takes the roots of the old and new PSI as input, compares them and creates a specification of text changes (6).

Fig. 3 describes the subsystem for generating interfaces and classes for PSI tree access. It consists of the following components:

- Specification Processor is responsible for processing and validating the pre-written developer specification of the syntax construct types of the programming language for which the PSI is being built.
- Scheme Manager stores knowledge about the schemes for generating Java interfaces and classes created based on type information from Specification Processor: which interfaces are implemented, the order of children during generation, and so on. Scheme Manager implements the Singleton design pattern.
- Types Manager stores knowledge about the semantic of syntax construct types: types of children, properties, etc. Similarly to Scheme Manager, it implements the Singleton design pattern.
- Generation is the main component that contains everything related to PSI generation. It provides the Generator interface, which is responsible for generating a specific file.

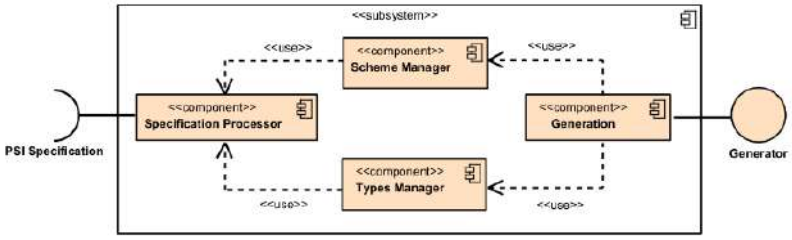


Fig. 3. The subsystem for generating interfaces and classes for PSI tree access

As a result of using the Writable PSI Generator, a user who wants to generate PSI for their IDE only needs to write a specification of types of syntax constructs for the corresponding programming language and run the generator. If necessary, Writable PSI Generator can be extended to take into account the specifics of a particular language.

3. Implementation Details

This section discusses the features and implementation details of the components described in Section 2.

3.1 PSI Modification Component

As mentioned in Section 2, the PSI Modification component is responsible for modifying and creating new PSI nodes. It also provides the functionality to completely rewrite the entire file tree.

PSI is a Lossless Syntax Tree (LST) [3], i.e., it stores information about whitespaces and comments in special nodes called Trivia and every PSI node stores its source text position and length.

Fig. 4 shows an example of a Lossless Syntax Tree for a simple Java return statement.

We chose Persistent Tree as the main approach for building the PSI and its modification system. Here, persistence means that when a data structure is modified, a new version of this data structure is returned. In addition, the unchanged parts of the data structure are reused. This approach provides the following benefits.

- Thread-safety via PSI immutability, as it eliminates the need for synchronization. In the IntelliJ Platform, for example, it is necessary to use Read and Write Action to interact with the PSI because of tree mutability [6].
- Fixed offsets and lengths of nodes in the tree. Immutability makes it possible not to be concerned about updating node offset in the text, as it will be correct after recreating the node.

- Secure manipulation of semantic information via separating the stages of tree modification. The user clearly knows when the semantic information is relevant.

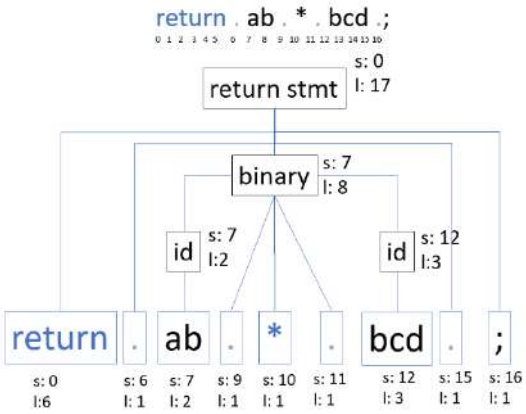


Fig. 4. Lossless Syntax Tree example

In constructing this data structure, we opted for the Red-Green-Trees method from Microsoft Roslyn [4]. This approach results in PSI constructed as a combination of two trees. The Green tree is an immutable untyped tree built during parsing. Its nodes (green) store information about node length in text, type, etc. They also store references to their children, but not to their parents. This tree is an implementation detail, and it is kept hidden from clients. The Red tree is an immutable typed tree, which is built lazily on demand from top to bottom. This is the PSI that the client works with. The nodes of this tree (red) reference the corresponding green nodes. Each red node stores an offset in the text document and a reference to its parent.

This approach to PSI construction resulted in a correctly working persistent data structure. The two trees are needed to provide the ability to iterate over the parents and children of PSI nodes.

Fig. 5 shows a simple example of this approach.

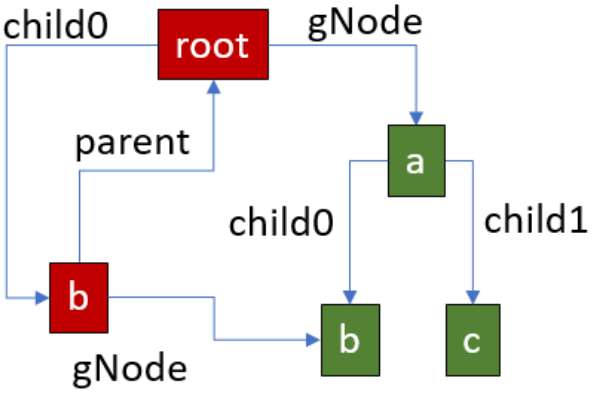


Fig. 5. An example of the Red Nodes and Green Nodes approach

In order to reduce the number of errors when working with the developed subsystem, we proposed an identifier system where each green node is assigned an identifier. This identifier is transferred to the new version when modifying and creating a new node. Modification methods and factory methods take these identifiers into account, which made it possible to build a more convenient API and support more PSI Modification Component usage scenarios.

As mentioned earlier, PSI stores Trivia nodes with whitespaces and comments. Microsoft Roslyn maintains the invariant that a node is Trivia if and only if it is a child of a token. This invariant is convenient for compiler system development, because it eliminates the problem of space and comment placement, leaving it as the responsibility of the client.

This approach was not applicable in the context of our IDE platform due to complicated API and difficulties in developing external services, and therefore Trivia nodes were placed in a more classical way — at the token level. The PSI Modification Component is responsible for whitespace normalization itself during node modification. *Fig. 6* (left) illustrates Trivia node placement in Microsoft Roslyn, while *Fig. 6* (right) shows the same for the Writable PSI Generator.

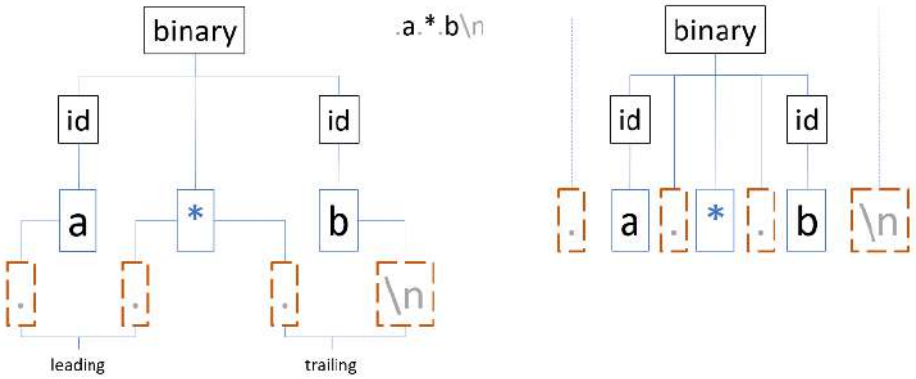


Fig. 6. Trivia nodes in Microsoft Roslyn and Writable PSI Generator

Modification Methods and Node Factory, which were mentioned in Section 2, are based on a common system for green node manipulation. Every Node Factory method makes it possible to build a new green node from other existing green nodes. A new red node is created based on the new green node. Every modification method uses the Node Factory method to build a new node based on the existing children. These methods are uniform and easy to generate. Furthermore, they are built based on the information about the syntax constructs of the language described in the specification, and thus produce only syntax-correct nodes.

The considered component also provides Tree Rewriter, an object that allows to replace or delete nodes in the PSI of the whole file. The replace and rewrite methods let Rewriter accumulate information about what changes should be applied to the tree. This is done by filling in the replace map and remove list, which store the data about the accumulated changes. The rewrite method activates PSI traversal, during which Rewriter replaces or removes nodes. This traversal is a Preorder Traversal, where the node itself is processed first, followed by its children from left to right. Rewriter takes into account the node offsets in source text, and therefore it does not have to traverse the whole tree. Instead, it only traverses the parts which have something to transform.

The result of the traversal is a new PSI. Rewriter takes into account the syntax structure of language constructions described in the specification, and does not produce a PSI with syntax errors.

As a result, the PSI Modification component meets the functional requirements described in Section 2, and achieves the following.

- Thread-safety.
- Syntax correctness after PSI transformation.
- Generatable API for modifying PSI nodes and producing new ones.
- Possibility to safely transform the PSI of an entire file.

3.2. Program Text Modification Component

As mentioned in Section 2, the Program Text Modification component is responsible for creating the shortest sequence of text changes that can be applied to the source document.

Three types of text changes are implemented: text insertion, text deletion, text replacement.

Each text change consists of the beginning and the end of the change, and the text that the document fragment needs to be replaced with. In the case of insertion, the beginning of the change is equal to the end. In the case of deletion, the string of text is empty.

This structure of text changes is due to the specifics of the IDE platform, for which the Writable PSI Generator was implemented.

In implementing the component, we decided to follow the GumTree approach [5], which produces text changes in two stages. First, it establishes the mappings between the nodes of the initial and final trees. Then, it analyzes these mappings and constructs a sequence of text changes based on the analysis.

GumTree made it possible to implement Program Text Modification, which generates the text sequence accurately and quickly. However, this approach required adaptation to the specifics of the developed IDE. Thus, GumTree allows to generate changes to move subtrees, which are not supported by the IDE. Therefore, these changes have been replaced by appropriate deletions and insertions. Also, text changes in the IDE platform are not applied sequentially — they are applied simultaneously. The approach has been adapted so that the created text changes meet the requirements of the platform. For example, multiple additions in a sequence in the same area are merged into a single change.

Such corrections allowed not only to adapt the approach to the requirements of the developed platform, but also to make them more convenient and less confusing, which was important when debugging the developed external services.

The implemented component is designed so that it can be applied to PSIs of different languages. The component itself has no knowledge of which programming language's trees it is analyzing.

3.3 Generation Subsystem

As it was mentioned in Section 2, the Generation Subsystem provides the ability to generate interfaces and classes to work with the PSI Modification component.

This subsystem is based on a given specification. The JSON format was chosen since it is widespread and has convenient processing and generation tools.

The specification contains the information necessary both for the operation of the generator and for the correct functioning of the entire modification subsystem. First of all, it contains definition of PSI node types according to programming language syntax. It describes what kind of children the PSI node can have according to the grammar of the language. Modification methods and factory methods are generated based on this information. On the second turn, it contains additional information for the generator. For example, it can specify if the class generated for a given type should be abstract, or if a factory method should be generated for a particular PSI node type, among others.

The specification is processed inseveral steps.

- Parsing and validation of the specification file
- Initialization of the Types Manager component based on the result of the first step
- Initialization of Scheme Manager component based on the result of the first step

During these steps, the specification file is validated to prevent unexpected system behavior. Checks for JSON object's mandatory and optional attributes, their types and values are performed.

The initialized Types Manager and Scheme Manager are objects that implement the Singleton pattern. They are available to both the generator and the PSI Modification component.

The generator is based on a Java StringBuilder, which builds a string that is the content of the generated file based on information from Types Manager and Scheme Manager. This string contains the package name, imports, fields, constructors, methods, etc., and it is written to the desired file. Such generation approach appeared to be the most suitable in the context of the IDE platform due to its simplicity and sufficient flexibility.

The described generation approach addressed another problem as well. Typically, generators produce files that prohibit manual code additions, because repeated generation of additional text is overwritten. However, the generator that we developed can create areas where code is not overwritten and it is possible to add new logic. This is done as follows. The generator checks if the file exists on disk. If the file does not exist, it is generated. The code is partitioned, leading to the division of the file into areas. Within one group of these areas, the code cannot be re-generated (e.g., the zone of generated methods, the zone of generated fields, etc.). The user can write code in these areas, and they are not re-generated. If the file exists, the generator recognizes via special area markers where the re-generation should be performed. The re-generated areas are replaced by new ones, the rest remain unchanged. The updated file is obtained by concatenating the contents of the generated and non-generated areas. Thus, developers are able to implement additional logic in the generated interfaces and classes. *Fig. 7* illustrates how a Java interface is derived from the type specification of syntactic constructs. This figure also showcases the division of code into areas where generation does or does not take place.

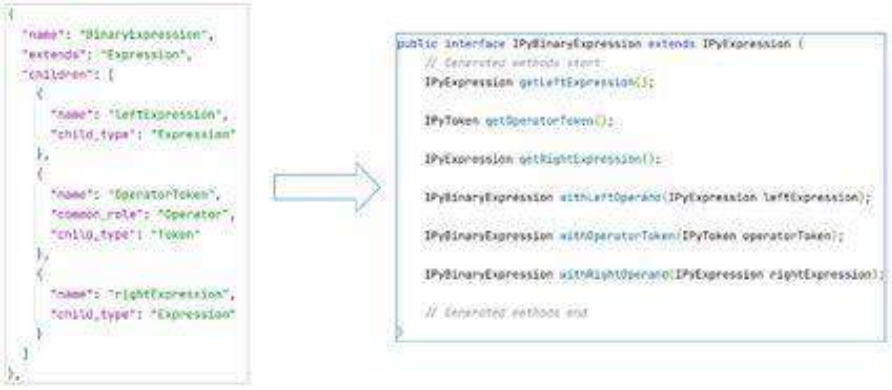


Fig. 7. Example of interface generation based on a JSON specification

4. Use Cases

This section shows how versatile and convenient the Writable PSI Generator is. The Java IDE and Python IDE are a software product line developed on the basis of a multi-language platform and its reusable assets [7]. The Writable PSI Generator presented in the paper is one of the reusable assets of the platform.

During the usage of the system by different products of the product line, it was improved: errors were corrected and new features were added. In terms of paper [8], this process is called improvement of reusable assets.

Specifications of syntax construct types for Python and Java languages were created, and based on these specifications, interfaces and classes for Python/Java PSI and other auxiliary code were generated. As a result, 21/7 improvements and bug fixes were made to the Writable PSI Generator in response to requests from the Python/Java teams.

It can be seen that the number of requests for such improvements when using the Writable PSI Generator decreased from product to product, indicating successful reuse of the asset.

4.1 Using the system within the Python IDE

Based on Writable PSI Generator for Python, the following features were implemented for the Python IDE.

- *Rearrange Code* is a refactoring that rearranges program constructs in source code. For example, this feature enables the developer to quickly move selected functions and classes through the source code, and reorder function arguments. It can also move functions out of classes into the external scope if the function is either at the very top or the very bottom of the class.
- *Introduce Variable* is a refactoring which lets the developer define a new variable for a selected expression, to which it will be assigned.
- *MinMax If* is a quick fix that allows the developer to turn a construct of the form *if a<b: return a else return b* into *return min (a,b).* There is no such feature in PyCharm at the moment.
- *Annotated Assignment* is a quick fix that allows the user to remove type annotation in case of chain assignment. (For example, a: int = b = d = 3 turns into a = b = d = 3). Python does not allow for type annotations in case of chain assignment: this is a syntax error. However, at the time of development of the Writable PSI Generator, even though PyCharm indicated an error in this case, it did not offer a quick fix.

4.2 Using the system within the Java IDE

Using the Writable PSI Generator for Java, the following features were implemented in the Java IDE.

- *Rename Method* is a refactoring that allows the developer to rename a class method and all its uses within the project.
- *Remove Useless Statement* is a quick fix that removes a useless construct in the source code (for example, an empty if statement).
- *Simplify Trivial If* is a quick fix that replaces an if statement with a return of *true* or *false* depending on a condition with a return with a check of this condition.

5. Related Work

PSI was first introduced in [2] for describing the syntax and semantic information of the developed program in IDE. However, the authors presented only general ideas regarding operation with PSI, without considering major non-trivial tasks associated with the PSI, such as tree modification and program text changing.

5.1 PSI Modification

Study [9] outlines the problem of refactoring service development and describes approaches to building a tree that is more convenient for IDEs. It highlights that in the IDE context the tree should store spaces and comments, and it should also be able to store positions and lengths of nodes in the text. Consequently, such a tree should be a Lossless Syntax Tree (LST), i.e. a tree which can be fully mapped to the original source code. However, this paper presents only a general view of the problem.

Paper [3] reviews different approaches to PSI design suitable for code refactoring services. It discusses two main approaches: Mutable Tree and Immutable Tree.

Mutable Tree is an approach in which tree nodes can be easily deleted, added or changed. It is quite appealing due to its simple implementation and convenient API, and therefore it is used in tools like IntelliJ Platform [6], Smalltalk Refactoring Browser [10], and CRefactory [11]. However, this approach has many disadvantages, such as problems with updating node offsets and lengths, and the need for synchronization in multi-threaded code.

Immutable Tree is an approach in which the tree cannot be altered once it is created. Paper [3] highlighted two approaches to designing a modification process on such a data structure: Rewriter and Persistent Tree. *Rewriter* is an approach in which all transformations over the tree are delegated to a separate object called Rewriter. The tree itself is not writable. This approach is employed in Eclipse Java Development Tools (JDT) and C/C++ Development Tools (CDT), addressing many of Mutable Tree problems, such as the lack of thread-safety. However, it does not provide an ability to interact with intermediate and final versions of trees during the modification process. *Persistent Tree* is an approach which allows clients to execute transformation actions on the tree. However, with every such operation, they receive an updated version of the tree, reflecting the applied transformations. This approach is used in the Microsoft Roslyn compiler written in C#. Its creators describe [4] its implementation via Red-Green Trees, as described above. This method of PSI construction offers an API through the red tree and hides implementation details in the inner green tree. This approach has all the benefits of an immutable tree, but also provides a more convenient way of interacting with the tree to transform it. A notable disadvantage of this approach is the difficulty of creating a convenient API for clients, which is due to the non-trivial organization of the data structure of Persistent Tree.

5.2 Program Text Changing

After performing transformations on PSI, it is necessary to transfer the changes to the source document (the program's source code) in order to make them visible to the IDE user (developer). In this case, a large number of fine-grained program changes can lead to performance issues. This problem is known as the problem of obtaining the shortest sequence of text changes that can be applied to the source document. It reduces to the Tree Differencing problem, which has proven itself to be a long-term research topic.

A set of approaches for Tree Differencing with retrieving text changes for adding, deleting and updating nodes in PSI is described in [12]. The RTED algorithm [13] stands out from this set, but its asymptotic performance is insufficient to meet the requirements of our IDE.

Further work tries not only to improve the asymptotic performance, but also determine the moves of subtrees. This is important because many refactoring services are often reduced to this type of tree operations (e.g., the Rearrange Code refactoring). Paper [14] proposes an algorithm for tree differencing of LaTeX trees. It is better compared to previous approaches, and has good asymptotic behavior, but it also has a significant limitation: the algorithm operates on trees that have a large amount of text in the leaves, which is not true for IDEs.

ChangeDistiller [15] improves on the ideas proposed earlier and makes the approach from [14] more suitable for Abstract Syntax Trees (ASTs). While it improves the asymptotic behavior, it still does not address the aforementioned limitation.

However, this limitation is solved by the GumTree algorithm described in [5]. It has suitable asymptotic performance for the needs of our IDE and is the most suitable for PSI differencing. Moreover, it generates a reasonably accurate sequence of textual changes, which also includes operations for moving subtrees. The disadvantage of this approach is that it can generate confusing textual changes, as discussed in [16]. This can be important when debugging an external service.

Paper [17] attempts to fix this problem by providing improvements for GumTree, which increases the accuracy of textual changes. However, this approach shows the best results only with Java code, and, consequently, is not well suited for a multilanguage platform.

6. Conclusion

The use of the Writable PSI Generator in IDE development projects for Python and Java has significantly improved the efficiency of PSI development by generating a significant amount of code and reducing the number of errors. Positive feedback has been received from the developers.

It should also be noted that the first product that used the Writable PSI Generator was the Python IDE, and the number of change requests within this implementation is larger than that for the next one. This suggests that improvement of reusable assets took place, which corresponds to the statements made in [7].

As a further direction of our work, we can specify the replacement of JSON for describing the types of programming language syntax constructs with a grammar-like language, for example, EBNF [18].

References

[1]. Alfred V. Aho, Ravi. Sethi, Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. — 1986. — P. 69–70.

[2]. Z. Kurbatova, Y. Golubev, V. Kovalenko and T. Bryksin. The IntelliJ Platform: A Framework for Building Plugins and Mining Software Data. 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Melbourne, Australia, 2021. — P. 14–17.

[3]. Jeffrey L. Overbey. 2013. Immutable source-mapped abstract syntax tree: a design pattern for refactoring engine APIs. In Proceedings of the 20th Conference on Pattern Languages of Programs (PLoP '13). The Hillside Group, USA, Article 7. — P. 1–8.

[4]. Lippert E. Fabulous adventures in coding. Blog. https://ericlippert.com/2012/06/08/red-green-trees/

[5]. Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14). Association for Computing Machinery, New York, NY, USA. — P. 313–324.

[6]. IntelliJ Platform SDK — Modifying the PSI. — URL: https://plugins.jetbrains.com/docs/intellij/modifying-psi.html.

[7]. A Framework for Software Product Line Practice, version 5.0, Software Engineering Institute, 2012, https://resources.sei.cmu.edu/

[8]. Попова Т.Н., Кознов Д.В., Тинова А.А., Романовский К.Ю. "Эволюция общих активов в семействе средств реинжиниринга программного обеспечения" // Системное программирование, 1 (2005), 184-198.

[9]. Peter Sommerlad, Guido Zgraggen, Thomas Corbat, Lukas Felber. Retaining comments when refactoring code — 2008. — 01. — P. 653–662.

[10]. Roberts Don, Brant John, Johnson Ralph. A Refactoring Tool for Smalltalk // TAPOS. — 1997. — 01. — Vol. 3. — P. 253–263.

[11]. Garrido Alejandra. Program Refactoring in the Presence of Preprocessor Directives: Ph. D. thesis / Alejandra Garrido. — USA: University of Illinois at Urbana-Champaign, 2005. — AAI3199001.

[12]. Bille Philip. A survey on tree edit distance and related problems // Theoretical Computer Science. — 2005. — Vol. 337, no. 1. — P. 217–239.

[13]. Pawlik Mateusz, Augsten Nikolaus. RTED: A Robust Algorithm for the Tree Edit Distance. — 2011. — 1201.0230.

[14]. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, Jennifer Widom. Change Detection in Hierarchically Structured Information. Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data. — SIGMOD ’96. — New York, NY, USA: Association for Computing Machinery, 1996. — P. 493–504.

[15]. Beat Fluri, Michael Wursch, Martin PInzger, Harald Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction / // IEEE Transactions on Software Engineering. — 2007. — Vol. 33, no. 11. — P. 725–743.

[16]. Guillermo de la Torre, Romain Robbes, and Alexandre Bergel. 2018. Imprecisions diagnostic in source code deltas. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18). Association for Computing Machinery, New York, NY, USA, 492–502.

[17]. J. Matsumoto, Y. Higo and S. Kusumoto. Beyond GumTree: A Hybrid Approach to Generate Edit Scripts. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 2019. — P. 550-554.

[18]. Pattis, Richard E. EBNF: A Notation to Describe Syntax. ICS.UCI.edu. University of California, Irvine.

Information about authors

Александр Сергеевич БОЖНЮК – младший инженер компании МПГ Айти Солюшнз. Закончил Математико-механический факультет СПбГУ по направлению «Программная инженерия». В сферу научных и профессиональных интересов входит разработка инструментов статического анализа и рефакторингов кода, теория компиляции, теория виртуальных машины, энергопотребление Android-устройств.

Alexander Sergeevich BOZHNYUK is and junior engineer of MPG IT Solutions. Graduated from the Faculty of Mathematics and Mechanics, St. Petersburg State University. His research and professional interests include development of static analysis and code refactoring tools, compilation theory, virtual machine theory, power consumption of Android devices.

Александр Александрович ЗАХАРОВ - инженер ключевых проектов компании МПГ АйТи Солюшнз, закончил Тульский государственный университет в 2005 году по направлению "Системный анализ и управление", профессиональные и научные интересы связаны с языками программирования, теорией и практикой инструментов для статического анализа и рефакторинга кода, теорией типов и реконструкцией типов.

Alexander Alexandrovich ZAKHAROV is an engineer of key projects of MPG IT Solutions, graduated from Tula State University in 2005 with a degree in "System Analysis and Management", professional and scientific interests are related to programming languages, theory and practice of tools for static analysis and code refactoring, type theory and type reconstruction.

Николай Владимирович ТРОПИН - ведущий инженер команды IDE в крупной телекоммуникационной компании, закончил Математико-механический факультет СПбГУ. Работает в области создания инструментов разработки с 2013 года.

Nikolay Vladimirovich TROPIN is the leading engineer of the IDE team in a large telecommunications company, graduated from the Mathematics and Mechanics Faculty of St. Petersburg State University. He has been working in the field of development tools since 2013.

Михаил Валериевич ВОЛКОВ, к.ф.-м.н., закончил физический факультет СПбГУ и аспирантуру Стокгольмского Университета. Занимался исследованиями в области квантовой механики. В данный момент профессиональные и научные интересы лежат в области разработки IDE, кодовых моделей и вывода типов для динамически типизированных языков.

Mikhail Valeryevich VOLKOV, PhD in Physics, graduated from the Physics Department of St. Petersburg State University and completed his postgraduate studies at Stockholm University. He was engaged in research in the field of quantum mechanics. At the moment his professional and scientific interests are in the field of IDE development, code model and type inference for dynamically typed languages.

DOI: 10.15514/ISPRAS-2023-35(3)-3

К вопросу использования в образовательном процессе разработанного программного обеспечения для исследования акустических трактов магнитострикционных преобразователей перемещений

А.А. Ивзанов, ORCID: 0009-0009-9664-4137 <*devilkbmod@gmail.com*>
А.А. Воронцов, ORCID: 0000-0002-3451-8815 <*aleksander.vorontsov@gmail.com*>
Ю.Н. Слесарев, ORCID: 0000-0003-1508-3235 <*slesarevun@gmail.com*>

Пензенский государственный технологический университет,
Россия, 440039, г. Пенза, проезд Байдукова/ул. Гагарина, 1а/11.

Аннотация. Статья посвящена исследованиям процессов, возникающих при формировании, трансляции и считывании информационных сигналов в акустических трактах магнитострикционных преобразователях линейных и угловых перемещений. Приводятся математические модели, позволяющие осуществить расчет магнитных полей кольцевых постоянных магнитов и сформированных токовыми импульсами при протекании ими в среде волновода. Для расчета намагниченности волновода был разработан численный метод, позволяющий учесть намагниченность материала волновода в предыдущий момент времени. Также приведены математические модели, позволяющие рассчитать параметры магнитного потока соленоида и выходного сигнала. Математические модели расчета магнитных полей постоянного магнита, разработанный численный метод и математические модели формирования магнитного потока и выходного сигнала были реализованы в разработанном программном обеспечении, используемом в образовательном процессе. Результаты исследований, а также уточненные и разработанные методы расчета магнитных полей и численный метод могут использоваться для исследований магнитострикционных приборов как на этапе их проектирования, так и их эксплуатации, что снижает их конечную стоимость. Также необходимо отметить, что в статье не рассмотрены вопросы, связанные с обработкой выходного сигнала, что предоставляет возможности для дальнейших исследований и дальнейшей модификации программного обеспечения.

Ключевые слова: акустический тракт; магнитострикция; магнитострикционный прибор; программное обеспечение; образовательный процесс.

Для цитирования: Ивзанов А.А., Воронцов А.А., Слесарев Ю.Н. К вопросу использования в образовательном процессе разработанного программного обеспечения для исследования акустических трактов магнитострикционных преобразователей перемещений. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 47–62. DOI: 10.15514/ISPRAS–2023–35(3)–3

On the Issue of Using the Developed Software in the Educational Process for the Study of Acoustic Paths of Magnetostrictive Displacement Transducers

A.A. Ivzanov, ORCID: 0009-0009-9664-4137 <*devilkbmod@gmail.com*>
A.A. Vorontsov, ORCID: 0000-0002-3451-8815 <*aleksander.vorontsov@gmail.com*>
Yu.N. Slesarev, ORCID: 0000-0003-1508-3235 <*slesarevun@gmail.com*>

Penza State Technological University,
1a/11, Baydukova passage/st. Gagarina, Penza, 440039, Russia.

Abstract. The article is devoted to the research of the processes arising during the formation, translation and reading of information signals in the acoustic paths of magnetostrictive linear and angular displacement transducers. Mathematical models are given that make it possible to calculate the magnetic fields of annular permanent magnets and those formed by current pulses when they flow in a waveguide medium. To calculate the magnetization of the waveguide, a numerical method was developed that allows taking into account the magnetization of the waveguide material at a previous time. Mathematical models are also given to calculate the parameters of the magnetic flux of the solenoid and the output signal. Mathematical models for calculating permanent magnet magnetic fields, the developed numerical method and mathematical models for the formation of magnetic flux and output signal were implemented in the developed software used in the educational process. The research results, as well as refined and developed methods for calculating magnetic fields and the numerical method can be used to study magnetostrictive devices both at the stage of their design and operation, which reduces their final cost. It should also be noted that the article does not address issues related to the processing of the output signal, which provides opportunities for further research and further modification of the software.

Keywords: acoustic tract; magnetostriction; magnetostrictive device; software; educational process.

For citation: Ivzanov A.A., Vorontsov A.A., Slesarev Yu.N. On the issue of using the developed software for the study of acoustic paths of magnetostrictive displacement transducers in the educational process. *Trudy ISP RAN/Proc.ISP RAS*, vol. 35, issue 3, 2023. pp. 47-62 (in Russian). DOI: 10.15514/ISPRAS-2023-35(3)-3

1 Введение

Динамика исследований и открытий в сфере научных изысканий и информационных технологий за последние годы позволила существенно расширить возможности для их дальнейшего изучения [1-12]. Это стало возможно благодаря успешной работе не только ученых и научных школ, но и исследованиям студентов и преподавателей [13-20].

Подготовка квалифицированных специалистов в образовательном процессе малоэффективна без наглядных методов обучения, позволяющих объяснить и продемонстрировать базовые основы, принципы работы и конструктивные особенности процессов и явлений.

Так, для изучения явления магнитострикции, а также процессов формирования, трансляции и считывания сигнала в акустических трактах магнитострикционных преобразователей перемещений можно использовать разработанное авторами публикации программное обеспечение, позволяющее не только понять принцип работы этих устройств, но и разрабатывать для них новые элементы и конструкции. Одной из особенностей программного обеспечения является реализация разработанного численного метода по расчету намагниченности материала волновода. Разработанные и уточненные математические модели позволяют рассчитать значения напряженностей магнитных полей постоянного магнита и созданным токовыми импульсами, магнитного потока и выходного сигнала, сформированного соленоидом с графическим отображением последнего. Теоретическим основам процессов, возникающих при трансляции сигнала в акустических трактах магнитострикционных преобразователях перемещений, а также вычислительным

экспериментам с использованием разработанного программного обеспечения и экспериментальной проверке полученных результатов и посвящена статья.

2 Теоретический анализ

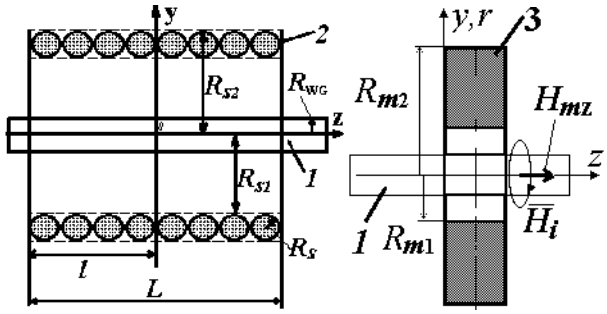
Принцип работы магнитострикционных приборов основывается на двух эффектах, заключающихся в изменении размеров материала волновода (ферромагнетик) под воздействием изменения его намагниченности, что приводит к изменению его размеров на локальном участке и формированию ультразвуковых волн (эффект Видемана) и обратному эффекту, называемому эффектом Виллари – изменении намагниченности материала волновода под воздействием механических колебаний (ультразвуковых волн).

Конструкция акустического тракта магнитострикционного преобразователя перемещений представлена на *Рис. 1*. В ней токовые импульсы *i* поступая в среду волновода (позиция 1) формируют магнитное поле, напряженность которого на его поверхности можно определить согласно выражению (1).

$$H_i = i/(2 \cdot \pi \cdot R_{WG}) \quad (1)$$

где R_{WG} – радиус волновода.

Магнитное поле, созданное токовыми импульсами, взаимодействует с магнитным полем постоянного магнита (позиция 3), что приводит к изменению намагниченности материала волновода и формированию механических колебаний, называемых ультразвуковыми волнами. Процесс расчета изменения намагниченности сложный и зависит от множества факторов и параметров. Более точно это изменение предлагается вычислить с использованием разработанного численного метода.



а) *б)*

Рис. 1. Поясняющий рисунок для дальнейших исследований. Изображение волновода (позиция 1) а) однослойного соленоида (позиция 2) и b) кольцевого постоянного магнита (позиция 3)

Fig. 1. An explanatory drawing for further research. Image of a waveguide (position 1) a) a single-layer solenoid (position 2), and b) an annular permanent magnet (position 3)

Ультразвуковые волны (акустический сигнал) распространяясь в среде волновода достигают соленоида (позиция 2), то есть элемента, формирующего выходной сигнал и считываются, после чего временной интервал их распространения преобразуется в измеряемое значения перемещения.

Для воспроизведения волны кручения может быть применен принцип взаимности. На *Рис. 1* показана катушка воспроизведения (соленоид), воспринимающая информативный параметр – продольную составляющую магнитного поля ультразвуковой волны кручения. На этом рисунке приняты следующие обозначения: R_S – радиус провода соленоида; $2l=L$ – длина соленоида; z – текущая координата, измеряемая вдоль оси 0Z; R_{S1} и R_{S2} – соответственно внутренний и внешний радиусы соленоида, $R_{S2}= R_{S1}+R_S$; R_{m1} и R_{m2} – внутренний и внешний радиусы кольцевого постоянного магнита соответственно.

Зная распределение поля соленоида $H_{CZ}(z)$, можно рассчитать значение проекции его магнитного потока на ось $0Z$ Φz через произвольное поперечное сечение волновода, вызванный током i через обмотку воспроизведения, так как в соответствии с теоремой взаимности тот же ток i, обтекая поперечное сечение поверхности волновода, возбуждает идентичный поток Φ через обмотку соленоида [1].

При движении волны кручения последовательность намагниченностей M_Z движется относительно соленоида. Следовательно, $M_Z=M(z-z_0)$, где $z_0=\upsilon t$, υ — скорость распространения ультразвуковой волны кручения. Переменная $(z-z_0)$ устанавливает среднюю линию центра неподвижного соленоида как начало системы отсчета для M_Z. Значение t (или z), равное нулю, соответствует определенной форме намагниченности, расположенной непосредственно под центром соленоида. Поле рассеяния соленоида $H_{col}(z)$, фиксируется по отношению к оси z.

Пусть величина поля рассеяния соленоида будет приведена к единичной магнитодвижущей силе (т. е. H определяется для $N1 = 1,0$, где N — число витков обмотки соленоида воспроизведения). Поле рассеяния соленоида воспроизведения, связанное с носителем, состоит из двух составляющих магнитного поля H_{cz} и H_{cy}. Таким образом, один ампер-виток обмотки воспроизведения будет возбуждать бесконечно малый поток через элемент волновода радиуса r и толщиной dy в декартовой системе координат или dr в цилиндрической системе координат определится согласно формуле (2).

$$d\Phi_Z=\mu_0 H_{CZ}\cdot 2\cdot\pi\cdot r\cdot dr \tag{2}$$

где $\mu_0=4\cdot\pi\cdot 10^{-7}$ Гн/м — магнитная постоянная.

Согласно теореме взаимности, единичный ток (ток через один виток вокруг элемента rdr), будет наводить аналогичный поток в обмотке соленоида воспроизведения.

Поскольку в волноводе ультразвуковой волны кручения распространяются со скоростью υ и горизонтальной составляющей намагниченности $M_z(z-\upsilon t)$ то магнитный момент элемента длиной dz и поперечного сечения $r\ dr$ эквивалентен току, текущему вокруг элемента rdr. Величина тока равна $M_z(z-\upsilon t)dz$. Этот эквивалентный ток представляет собой масштабный коэффициент, который должен быть использован в уравнении (3) для получения из последнего действительного уровня намагниченности волноводе.

Полный поток, пересекающий обмотку соленоида воспроизведения, вызванный намагниченностью M_z получается в этом случае посредством интегрирования бесконечно малых потоков, возникающих из последовательности намагниченностей вдоль волновода, и определяется согласно выражению (3).

$$\Phi_z(z_0)=k\cdot\int_{-\infty}^{\infty}\int_{0}^{\frac{d_{WG}}{2}}\mu_0\cdot M_z(z-z_0,r)\cdot H_{cr}(z,r)\cdot 2\pi r\cdot dr\cdot dz \tag{3}$$

где d_{WG} — диаметр волновода, $z_0=\upsilon t$, k — обобщенный коэффициент, включающий магнитострикционные коэффициенты.

Сигнал воспроизведения определяется производной от $\Phi_Z(z)$ (выражение (4)).

$$u(z_0)=\mathrm{d}\Phi_Z(z_0)/\mathrm{d}t \tag{4}$$

Намагниченность M_z определяется воздействием на материал волновода поля равного геометрической сумме кругового поля H_i волновода и горизонтальной составляющей поля постоянного магнита $Hz(r,z)$.

Для кольцевого постоянного магнита с намагниченностью M и с внешним и внутренним радиусами соответственно Rm_2 и Rm_1 значение напряженности магнитного поля $Hz(r,z)$ определится по формуле (5).

$$H_z(r,z)=H_{z2}(r,z)-H_{z1}(r,z) \tag{5}$$

где:

$$H_{z2}(r,z)=\frac{1}{\pi}\cdot\int_{Rm1}^{Rm2}\frac{M\cdot\rho\cdot z\cdot E2(k2)}{\left[(r-\rho)^2+z^2\right]\cdot\sqrt{(r+\rho)^2+z^2}}d\rho \tag{6}$$

$$H_{z1}(r,z)=\frac{1}{\pi}\cdot\int_{Rm1}^{Rm2}\frac{M\cdot\rho\cdot(z-h)\cdot E2(k1)}{\left[(r-\rho)^2+(z-h)^2\right]\sqrt{(r+\rho)^2+(z-h)^2}}d\rho \tag{7}$$

В соответствии с одной из моделей, поле H_{CZ} на оси соленоида в точке, отстоящей на расстоянии z от его центра, определяется по формуле (8).

$$H_{cz}(z)=\frac{ni}{4(Rs2-Rs1)}\left\{(l-z)\ln\frac{R_{s2}+\sqrt{R_{s2}^2+(l-z)^2}}{R_{s1}+\sqrt{R_{s1}^2+(l-z)^2}}+(l+z)\ln\frac{R_{s2}+\sqrt{R_{s2}^2+(l+z)^2}}{R_{s1}+\sqrt{R_{s1}^2+(l+z)^2}}\right\} \tag{8}$$

где $n=N/L$ — число витков на единицу длины соленоида, Rs_1 и Rs_2 – соответственно внутренний и внешний радиусы соленоида, $2l=L$ – длина соленоида, i – значение токового импульса, s – площадь поперечного сечения волноводе.

Используя выражение (5) можно получить распределение продольной намагниченности $M_Z(z)$ в волноводе, которое определяется функциональной зависимостью от $H_Z(r,z)$ то есть согласно выражению (9).

$$M_Z(z) = f\,(H_Z\,(r,z)) \tag{9}$$

Для моделирования намагниченности волновода также желательно аналитически определить зависимость, как это представлено в выражении (10).

$$M_i = M\,(H\,(x_i,t),\,M\,(H\,(x_i,t-\Delta t)), \tag{10}$$

где $H(x_i,t)$ – суммарное поле, действующее на магнитный элемент в момент времени t;

$M(H(x_i,t-\Delta t))$ – намагниченность элемента, определяемая в предшествующий момент времени $t-\Delta t$.

Вид этой функциональной зависимости определяется выбранной моделью намагничивания волноводе, моделями петли гистерезиса и многими другими параметрами.

Петля гистерезиса – одно из свойств ферромагнетиков (материал волновода). Оно позволяет определить его намагниченность по значению напряженности магнитного поля и наоборот. Существует множество моделей построения и методов аппроксимации петель Гистерезиса, одним из которых, используемым в разработанном программном обеспечении является полиномиальный метод аппроксимации [2] (выражение 11) модели петли гистерезиса по Нишимото [3].

$$M(H,\alpha_1)=M_s(T)\cdot\operatorname{sign}\alpha_1+M_s\cdot\alpha_1\cdot f(H\cdot\operatorname{sign}\alpha_1) \tag{11}$$

где Ti — температура переключаемого элемента, $f(H)$ — функция, описывающая предельную петлю гистерезиса и имеющая вид, приведенный в формуле (12) [3].

$$f(H)=\begin{cases}\dfrac{H_C-H}{H-\dfrac{H_C}{s_p}} & \text{при } H<H_C\\[2ex] 1-\left(\dfrac{H_C}{H}\right)^{\frac{s_p}{1-s_p}\cdot K_S} & \text{при } H>H_C\end{cases} \tag{12}$$

В данном описании петли гистерезиса $s_P=Mr/Ms$, K_S – коэффициент, меньший 1.

Графики, построенные по формулам (11), (12), приведены на *Рис. 2*.

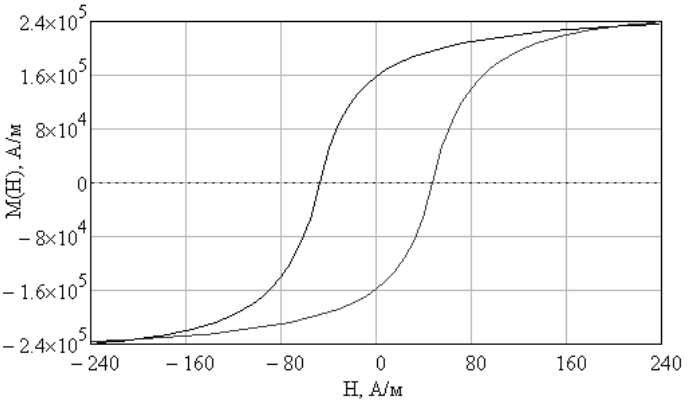


Рис. 2. Модель петли гистерезиса по Нишимото [3]: $H_C=150/\pi$ А/м, $M_S=33\cdot10^5/(4\cdot\pi)$ А/м, $s_p=0,6$, $M_r=s_p\cdot M_s$

Fig. 2. Nishimoto hysteresis loop model [3]: $H_C=150/\pi$ A/m, $M_S=33\cdot10^5/(4\cdot\pi)$ A/m, $s_p=0,6$, $M_r= s_p\cdot M_s$

Использование зависимости (9) производилось путем моделирования предельных и частных циклов петли гистерезиса [2,3] по формулам (10) и (11).

Индекс петли α_1 определяется из условий равенства величин намагниченностей в точке возврата [2] согласно выражению (13).

$$\alpha_1^{(k+1)} = \alpha_1^{(k)} \frac{M\left(H^{(k)}, \alpha_1^{(k)}\right) + M_S \cdot \operatorname{sign}\left(\alpha_1^{(k)}\right)}{M\left(H^{(k)}, -\alpha_1^{(k)}\right) + M_S \cdot \operatorname{sign}\left(\alpha_1^{(k)}\right)} \tag{13}$$

где k — порядковый номер цикла изменения намагниченности по петле гистерезиса.

Для вычисления по формулам (11) и (12) модель позволяет использоваться также температурные зависимости $H_c\,(x_i) = H_c\,(T_i\,(x_i))$ и $M(x_i) = M_S\,(T_i\,(x_i))$, получаемые путем интерполяции экспериментальных данных.

Общим недостатком методов расчета намагниченности волноводов по статическим петлям гистерезиса является невозможность учета в них влияния саморазмагничивающих полей. Их учет возможен при использовании разработанного численного метода.

Метод самосогласованного динамического моделирования (МСДМ) получил широкое распространение для моделирования записи информации на ферромагнитных носителях посредством приложения локального магнитного поля, создаваемого например магнитной головкой [4,5].

Некоторые общие положения данной методики могут быть применены и для моделирования намагничивания волноводов магнитострикционных приборов на ультразвуковых волнах кручения с учетом особенностей физики перемагничивания круговым полем, методик определения магнитных полей и моделей перемагничивания материала волновода.

МСДМ представляют собой модели и методы приближенного решения нелинейной задачи, которые для случая формирования намагниченности в волноводе имеют вид, представленный в формуле (14).

$$\begin{aligned} \overline{M}(\bar{r},t) &= \overline{m}_{tf}\left(\overline{H}_\Sigma(\bar{r},t), history\overline{H}_\Sigma\right) \\ \overline{H}_\Sigma(\bar{r},t) &= H_{\text{external}}(\bar{r},t) + H_{\text{demagnetizing}}(\bar{r},t), \end{aligned} \tag{14}$$

где $\overline{M}(\bar{r},t)$ – вектор намагниченности в точке с радиусом–вектором $\bar{r}$ в момент времени t; $\overline{H}_\Sigma(\bar{r},t)$ – вектор напряженности полного магнитного поля, включающий сумму внешнего

поля $H_{\text{external}}(\bar{r},t)$, создаваемого совместным действием постоянного магнита и циркулярного магнитного поля в отсутствие магнитного волновода, и внутреннего поля $H_{\text{demagnetiz ing}}(\bar{r},t)$, существующего в магнитном материале; $\overline{m}_{tf}$ – моделирующая функция, связывающая остаточную намагниченность волновода с напряженностью магнитного поля; $\overline{H}_{\text{demagnetiz ing}}(\bar{r},t)$ – размагничивающее поле, определяется из магнитостатических уравнений Максвелла согласно выражению (15).

$$\begin{aligned} &\text{div}\overline{H}_{\text{demagnetiz ing}}(\bar{r},t) = -4\cdot\pi\cdot\text{div}\overline{M}(\bar{r},t); \\ &H_{\text{demagnetiz ing}}(r\to\infty)\to 0. \end{aligned} \tag{15}$$

Различные МДСМ отличаются выбором моделирующей функции $\overline{m}_{tf}(\overline{H}, history\overline{H})$ и математическими методами приближенного решения уравнений (14), (15).

Следует отметить, что в традиционных моделях формирования намагниченности для описания процесса формирования распределения намагниченности использовались и различные модели образования доменов, где основой рассмотрения является минимизация термодинамического потенциала.

Между тем известно, что в большинстве магнитных материалов, в том числе и в волноводе, собственная доменная структура может иметь размеры гораздо меньшие, чем области, перемагничиваемые под действием магнитного поля. Если это так, то эти области можно описать при помощи параметров, характерных для макрообластей, в частности, коэрцитивной силой, коэффициентом прямоугольности петли гистерезиса, которые могут быть легко измерены и отражают структурное состояние реального материала. Использование этих параметров позволяет применить к рассмотрению явлений в волноводе некоторые подходы, развитые для моделирования магнитной записи магнитной головкой.

Задача расчета изменения намагниченности волновода при изменении магнитного поля решается следующим образом: (*Рис. 3*).

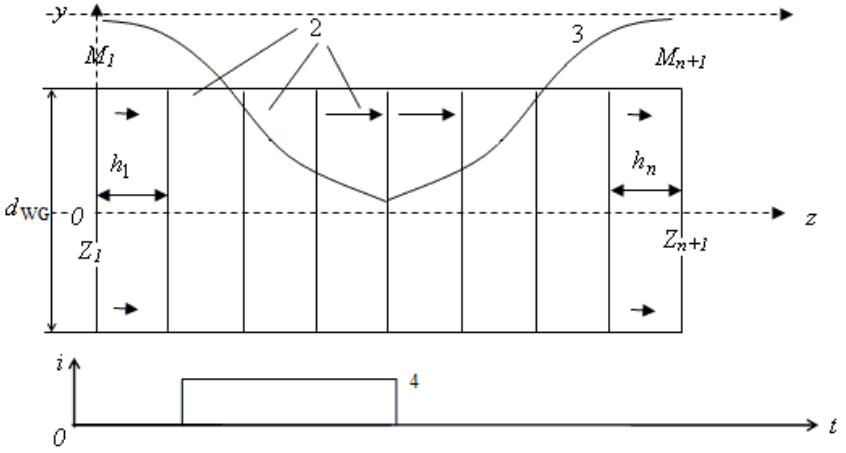


Рис. 3. Модель формирования намагниченности в волноводе, где 1— волновод, 2 — области разбиения волновода, 3 — распределение продольной составляющей результирующего магнитного поля, 4 — импульс тока в волноводе в зависимости от времени

Fig. 3. Model of magnetization formation in a waveguide, where 1 is a waveguide, 2 is a waveguide partition region, 3 is the distribution of the longitudinal component of the resulting magnetic field, 4 is a current pulse in the waveguide depending on time

Для определения намагниченности и магнитных полей при перемагничивании волновода рассматриваемая область волновода разбивается на *n* элементов длиной *h*, ограниченных

узловыми точками Z_i. В каждый момент времени можно задать поле температур $T_i=T(Z_i)$. При разработке модели принимаются следующие допущения:

1 Считается, что в каждом элементе разбиения намагниченность является постоянной величиной.

2 Соответственно каждый элемент разбиения будет представлять цилиндрическую область радиусом d_{WG} /2.

Практически распределение намагниченности численно вычисляется с помощью метода итерации. Обобщенная процедура итеративного расчета приведена на *Рис. 4*. Детализированный алгоритм разработанного численного метода приведен на *Рис. 5*.

Для описания намагниченности в предшествующий момент времени между узлами Z_i и Z_{i+1} применяется линейная интерполяция.

Суммарное магнитное поле $H_\Sigma(Z_i,t)$ определяется в соответствии с выражением(16).

$$H_\Sigma(Z_i, t) = H_{\text{external}}+\Sigma A_{i,j}\cdot M_j(t), \tag{16}$$

где $A_{i,j}$ – элемент матрицы форм-факторов, получаемой с учетом интерполяции намагниченности в элементарном магните путем интегрального решения магнитостатических уравнений Максвелла в форме скалярного потенциала; i — положение точки наблюдения (*Рис. 3*); j – положение границы конечного элемента с магнитным материалом.

Для расчетов суммарного магнитного поля $H_\Sigma(Z_i,t)$, описываемого выражением (16), возможно использование методов итерации, наиболее адаптированным под рассматриваемую модель из которых является метод релаксации. Данный метод решения систем алгебраических уравнений обладает большой скоростью сходимости благодаря тому, что в нем после вычисления очередной i–ой компоненты (k+1)-го приближения по формуле метода Зейделя производят дополнительное смещение этой компоненты.

$$H_{external} \rightarrow M_i^1 \rightarrow H_{\text{demagnetizing}}{}_i^1$$
$$\downarrow$$
$$(H_{external} + H_{\text{demagnetizing}}{}_i^1) \rightarrow M_i^2$$
$$\downarrow$$
$$B_i^2 \rightarrow H_{\text{demagnetizing}}{}_i^2$$
$$\downarrow$$
$$(H_{external} + H_{\text{demagnetizing}}{}_i^2) \rightarrow M_i^3$$
$$\downarrow$$

Рис. 4. Обобщенная процедура итеративного расчета намагниченности в волноводе
Fig. 4. Generalized procedure for iterative calculation of magnetization in a waveguide

Выбор данного метода был также осуществлен из-за возможности дополнительного введения в расчеты релаксации по индукции, что повышает устойчивость.

Согласно этому методу значение напряженности магнитного поля H определится в соответствии с выражением (17).

$$H^{(k)}=H^{(k-1)}+\lambda_1(H^{(k-1)}+A(B^{(k-1)} - H^{(k-1)})), \tag{17}$$

где λ_1 – некоторое число, называемое коэффициентом ускорения сходимости по напряженности магнитного поля, определяющее метод решения выражения (15). Так, при $\lambda_1<1$, формула (17) является формулой метода нижней релаксации, $\lambda_1=1$ – формулой Зейделя,

λ_1>1 – формулой верхней релаксации. $B^{(k-1)}$ – значение магнитной индукции, вычисленной для (k-1)-го шага итерации.

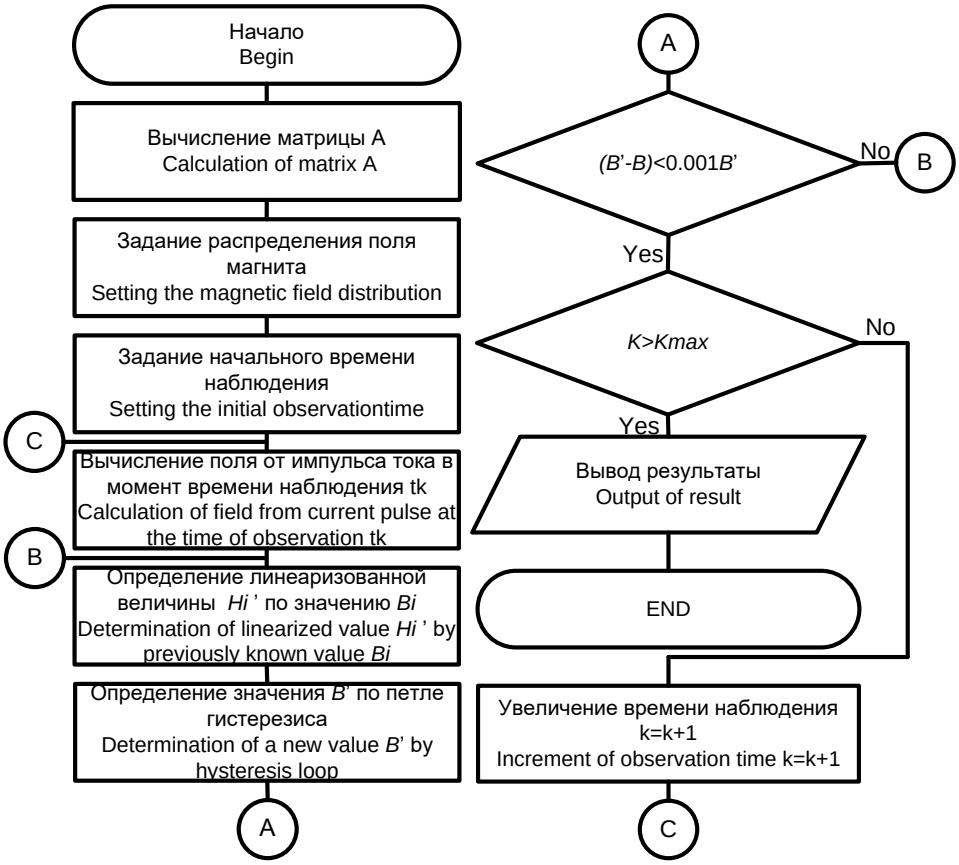


Рис. 5. Алгоритм работы разработанного численного метода
Fig. 5. Algorithm of the developed numerical method

Как уже отмечалось, для повышения устойчивости, в известный метод расчета MSDM дополнительно вводится релаксация по индукции, значение которой на k-ом шаге итерации можно будет определить согласно выражению (18).

$$\overline{B}^{(k)} = (1-\lambda_1^{'}) \cdot \overline{B}^{(k-1)} + \lambda_1^{'} \cdot \overline{B}(\overline{H}^{(k)}), \tag{18}$$

where $\overline{H}^{(k)}$ – значение напряженности магнитного поля на k-ом шаге итерации, определяемое согласно выражению (17), λ_1' – коэффициент ускорения сходимости по магнитной индукции.

Введение дополнительной релаксации по индукции повышает устойчивость рассматриваемого метода и является отличием предложенного численного метода от существующих.

Вид матрицы форм-факторов зависит от ориентации намагниченности относительно поверхности волновода, способа интерполяции намагниченности между узлами элементов разбиения.

Элементы матрицы форм-факторов определяются вкладом магнитного поля от областей волновода, лежащих вне его участков, ограниченных точками разбиения Z_1 и Z_{n+1}, а также элементами разбиения h_i.

Для определения величины размагничивающего поля используется следующее известное выражение [6].

$$H_{\text{demagnetizing}}(r) = \int(-\nabla M)\left[(\bar{r}-\bar{\rho})/\left|\bar{r}-\bar{\rho}\right|^3\right]dV + \int \bar{M}\cdot\bar{n}\left[(\bar{r}-\bar{\rho})/\left|\bar{r}-\bar{\rho}\right|^3\right]ds \qquad (19)$$

где $\bar{r}$ – радиус–вектор точки наблюдения; $\bar{\rho}$ – радиус–вектор, проведенный к магнитному материалу; n – единичная нормаль к боковой поверхности элемента разбиения; V,s – соответственно объем и поверхность магнетика; ∇M – оператор дивергенции намагниченности.

Интегрирование (19) позволяет определить поле от первого элемента разбиения в соответствии с выражением (20).

$$H_1(r,z) = \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_1\cdot(z-h)\cdot E(k_1)\rho\cdot d\rho}{[(r-\rho)^2+(z-h)^2]}\times\frac{1}{[(r+\rho)^2+(z-h)^2]^{\frac{1}{2}}}\right] - \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\frac{M_1\cdot z\cdot E(k_2)\rho\cdot d\rho}{[(r-\rho)^2+z^2]\cdot[(r+\rho)^2+z^2]^{\frac{1}{2}}} \qquad (20)$$

где $E(k_2) = \int_0^{\frac{\pi}{2}}\sqrt{(1-k_2^2(\sin\varphi)^2)}d\varphi$ – полный эллиптический интеграл второго рода, $k_1^2 = (4\cdot r\cdot\rho)/\left[(r+\rho)^2+(z-h)^2\right]$;. $k_2^2 = (4\cdot r\cdot\rho)/\left[(r+\rho)^2+(z)^2\right]$; r – радиус точки наблюдения, отсчитываемый от выбранного начала координат, z – координата, параллельная направлению намагниченности волновода.

Аналогично интегрирование выражения (20) позволяет также определить соответственно поле от второго, третьего и т.п. элементов.

$$H_2(r,z) = \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_2\cdot(z-2\cdot h)\cdot E(k_1)\rho\cdot d\rho}{[(r-\rho)^2+(z-2\cdot h)^2]}\times\frac{1}{[(r+\rho)^2+(z-2\cdot h)^2]^{\frac{1}{2}}}\right] - \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_2\cdot(z-h)}{[(r-\rho)^2+(z-h)^2]}\times\frac{E(k_2)\rho\cdot d\rho}{[(r+\rho)^2+(z-h)^2]^{\frac{1}{2}}}\right]$$

$$H_3(r,z) = \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_3\cdot(z-3\cdot h)\cdot E(k_1)\rho\cdot d\rho}{[(r-\rho)^2+(z-3\cdot h)^2]}\times\frac{1}{[(r+\rho)^2+(z-3\cdot h)^2]^{\frac{1}{2}}}\right] - \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_3\cdot(z-h)^2}{[(r-\rho)^2+(z-2\cdot h)^2]}\times\frac{E(k_2)\rho\cdot d\rho}{[(r+\rho)^2+(z-2\cdot h)^2]^{\frac{1}{2}}}\right]$$

$$H_n(r,z) = \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_n\cdot(z-n\cdot h)\cdot E(k_1)\rho\cdot d\rho}{[(r-\rho)^2+(z-n\cdot h)^2]}\times\frac{1}{[(r+\rho)^2+(z-n\cdot h)^2]^{\frac{1}{2}}}\right] - \frac{1}{\pi}\cdot\int_0^{d_{WG}/2}\left[\frac{M_n\cdot(z-(n-1)\cdot h)}{[(r-\rho)^2+(z-(n-1)\cdot h)^2]}\times\frac{E(k_2)\rho\cdot d\rho}{[(r+\rho)^2+(z-(n-1)\cdot h)^2]^{\frac{1}{2}}}\right]$$

где $k_1^2 = (4\cdot r\cdot\rho)/\left[(r+\rho)^2+(z-n\cdot h)^2\right]$ и $k_2^2 = (4\cdot r\cdot\rho)/\left[(r+\rho)^2+(z-(n-1)\cdot h)^2\right]$.

При $r=d_{WG}/2$ и $z=z_i=h\cdot$i можно определить коэффициенты матрицы форм-факторов $A_{i,j}$ по формуле (21).

$$A_{i,j}=\frac{1}{\pi}\cdot\int_0^{d_{\mathrm{WG}}/2}\left[\frac{(i\cdot h-j\cdot h)E(k_1)\rho\cdot d\rho}{[(\frac{d_{\mathrm{WG}}}{2}-\rho)^2+(i\cdot h-j\cdot h)^2][(\frac{d_{\mathrm{WG}}}{2}+\rho)^2+(i\cdot h-j\cdot h)^2]^{\frac{1}{2}}}\right]-$$
$$-\frac{1}{\pi}\cdot\int_0^{d_{\mathrm{WG}}/2}\left[\frac{(i\cdot h-(j-1)\cdot h)E(k_2)\rho\cdot d\rho}{[(\frac{d_{\mathrm{WG}}}{2}-\rho)^2+(i\cdot h-(j-1)\cdot h)^2][(\frac{d_{\mathrm{WG}}}{2}+\rho)^2+(i\cdot h-(j-1)\cdot h)^2]^{\frac{1}{2}}}\right] \quad (21)$$

где $k_1^2=(2\cdot d_{\mathrm{WG}}\cdot\rho)/\left[(d_{\mathrm{WG}}/2+\rho)^2+(i\cdot h-j\cdot h)^2\right]$ и $k_2^2=(2\cdot d_{\mathrm{WG}}\cdot\rho)/\left[(d_{\mathrm{WG}}/2+\rho)^2+(i\cdot h-(j-1)\cdot h)^2\right]$.

Описанная методика позволяет с учетом влияния размагничивающих полей определить намагниченность волновода как на локальном участке, так и вдоль всей длины волновода.

Разработанный численный метод позволяет рассчитать значение намагниченности материала волновода, значение которого при подстановке в выражения (3) и (4) позволяет найти значения проекции на ось $0Z$ магнитного потока и выходного сигнала, сформированного соленоидом. Представленные в данном разделе математические модели и разработанный численный метод были реализованы в используемом в образовательном процессе программном обеспечении, о котором более подробно будет изложено в разделе, посвященном вычислительным экспериментам.

3 Вычислительный эксперимент

Используя уточненные и разработанные модели и численный метод, представленные в теоретической части, было разработано программное обеспечение, используемое в образовательном процессе, позволяющее произвести исследования в акустических трактах магнитострикционных преобразователей перемещений.

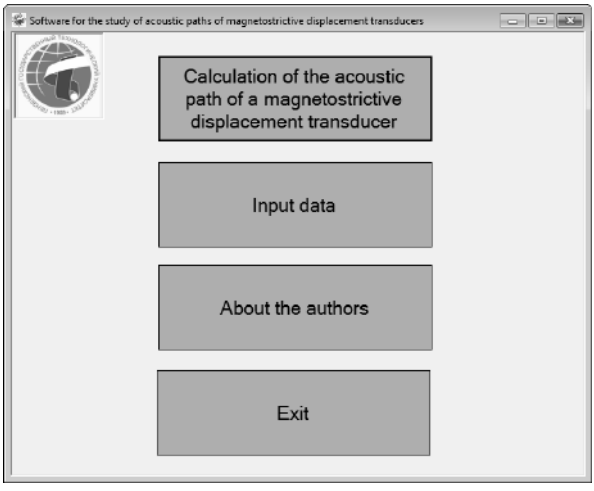


Рис. 6. Основная форма разработанного программного обеспечения
Fig. 6. The main form of the developed software

Оно состоит из 5 модулей, в том числе основного, содержащего меню и представленного на *Рис. 6*. При нажатии на кнопку “About the authors” появляется окно, показанное на *Рис. 7*.

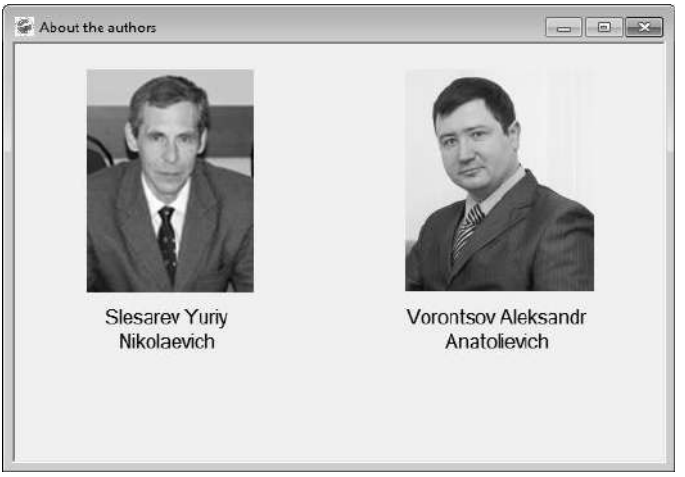


Рис. 7. Форма, содержащая информацию об авторах
Fig. 7. A form containing information about the authors

Форма “Input data” (*Рис. 8*) появляется при нажатии в основной форме соответствующей кнопки и позволяет сохранить в текстовый файл, извлечь из него и работать с информацией о основных параметрах и свойствах элементов конструкции акустического тракта магнитострикционного преобразователя линейных перемещений. В этой форме имеется возможность выбрать форму, марку и свойства постоянного магнита и ввести его параметры, ввести значение диаметра и свойств материала волновода, амплитуду токовых импульсов и их циклическую частоту колебаний, параметры и размеры соленоида, количество витков и коэффициент заполнения его обмотки. После заполнения этой формы появляется возможность расчета выходных данных либо с использованием кнопки “Calculate” либо нажатием верхней кнопки основной формы. Форма расчета выходных данных, экранная форма которой представлена на *Рис. 9* позволяет получить вычисленные программным обеспечением данные о свойствах постоянного магнита и волновода, а также сравнить полученное значение результирующей напряженности магнитного поля с ее минимальным и максимальным возможными значениями (проверить, находится ли это значение в рабочей области).

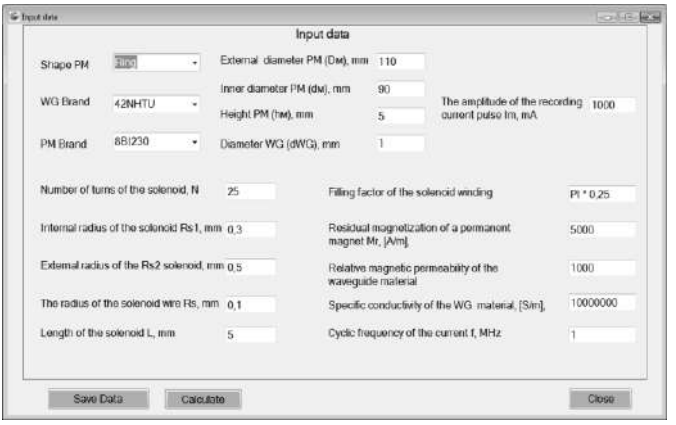


Рис. 8. Форма редактирования входных данных
Fig. 8. Input Data Editing Form

Также программное обеспечение может получить графическое представление выходного напряжения, сформированного соленоидом или передать полученные значения в MathCad для построения данных в этой системе.

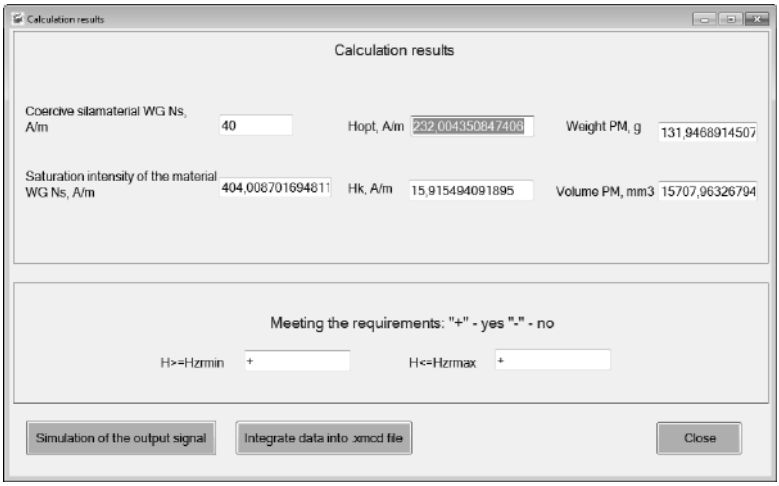


Рис. 9. Форма расчета выходных данных
Fig. 9. Output data calculation form

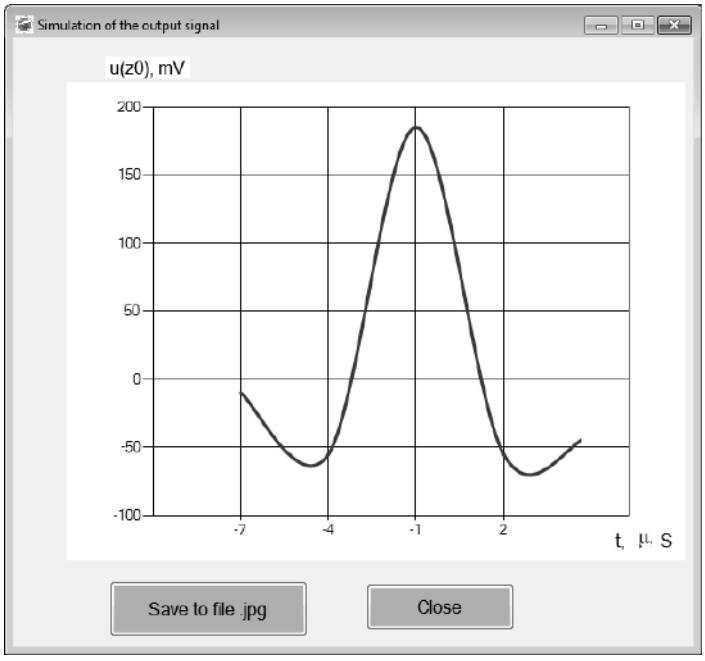


Рис. 10. Форма графического построения выходного сигнала
Fig. 10.The form of graphical construction of the output signal

Результаты одного из вычислительных экспериментов по исследованию выходного сигнала, сформированного соленоидом от времени с использованием разработанного программного обеспечения приведен на *Рис. 10*. Для проверки адекватности его отображения возможно проведение экспериментальной проверки, осуществляемой в экспериментальной части.

4 Экспериментальная часть

Для проверки адекватности вычислительных экспериментов была проведена экспериментальная проверка, с использованием лабораторного стенда (*Рис. 11*), собранного авторами статьи, содержащего генератор импульса треугольной формы, соленоид воспроизведения, преобразующий перемещающуюся крутильную волну в электрический сигнал, усилитель сигнала воспроизведения. Осциллограмма сигнала возбуждения и измерительного сигнала приведена на *Рис. 12*. Результаты экспериментальной проверки для значений, использованных в вычислительных экспериментах показали совпадение не только формы выходного сигнала, но и его длительности. Амплитудное значение выходного сигнала отличалось не более 10%, что обусловлено погрешностями в свойствах и параметрах кольцевого постоянного магнита, волновода и соленоида, а также несовершенством намотки и коммутации соленоида.

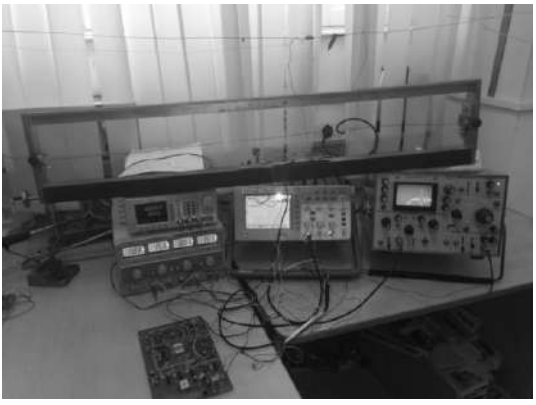
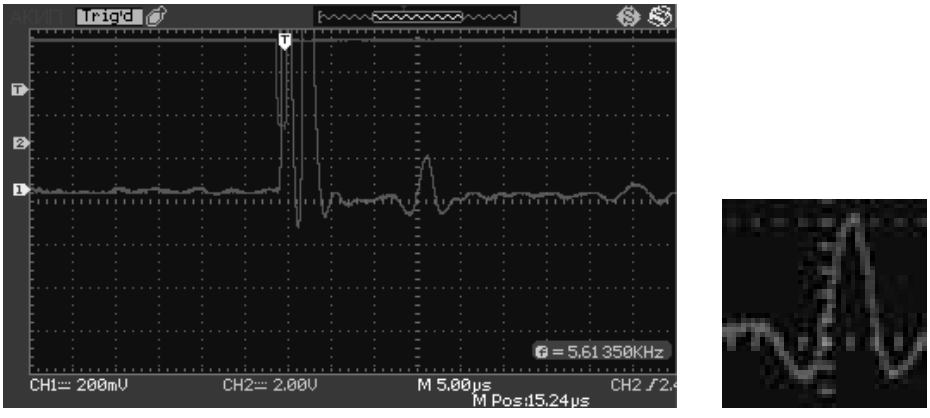


Рис. 11. Экспериментальная установка
Fig. 11. Experimental installation.=



а) *б)*

Рис. 12. Экспериментальная проверка по определению выходного сигнала
а) Полученный выходной сигнал и б) его масштабированная часть, содержащая выходной сигнал, сформированный токовым импульсом записи
Fig. 12. Experimental verification to determine the output signal. a) The received output signal and b) its scaled part containing the output signal generated by the recording current pulse

5 Заключение

Таким образом, в работе были уточнены и разработаны математические модели и численный метод, позволяющий проводить исследования в акустических трактах магнитострикционных преобразователях перемещений. Использование разработанного программного обеспечения в образовательном процессе позволит не только изучать основные явления и процессы, происходящие в акустических трактах магнитострикционных приборов, но и проектировать новые конструкции и подбирать необходимые габариты и свойства элементов конструкции, что представляет перспективы для разработки новых классов и конструкций магнитострикционных приборов. Результаты вычислительных экспериментов, полученных разработанным программным обеспечением совпали с экспериментальной проверкой с использованием экспериментальной установки, что подтверждает адекватность представленных моделей и разработанного численного метода.

Список литературы / References

[1]. Хогленд А. Цифровая магнитная запись. М.: Советское радио, 1967. 280 с. / Hoagland A. Digital Magnetic Recording. M.: Soviet radio, 1967. 280 P.

[2]. Not Just Another Self- Consistent Magnetic Recording Model/ G.G. Hughes, D.S. Bloomberg, V. Castelli, R. Hoffman// IEEE Trans. Magn.- 1981.- MAG-17, № 2.- P. 1192-1199.

[3]. Computer simulation of high- density multiple transition in magnetic disk recording/ K. Nichimoto, Y. Nagao, Y. Suganuma, H. Tonaka// IEEE Trans. Magn.- 1974.- MAG.10.- №3.- P. 769-775.

[4]. George D.J., King S. F., Carr A.R. A self-Consistent Calculation of the Magnetic Transition Recording on a Thin Film Disc.- IEEE Transaction on Magnetics, June, 1971, p.240-243.

[5]. Ивасаки С. Динамическая модель процесса магнитной записи/ С. Ивасаки, Т. Судзуки // Проблемы магнитной записи: Пер. с англ. /Под ред. В.Г. Корольков.- М.: Энергия, 1975.- С. 25-34. / Iwasaki S. Dynamic model of the magnetic recording process/ S. Iwasaki, T. Suzuki // Problems of magnetic recording: Translated from English /Edited by V.G. Korolkov.- M.: Energiya, 1975.- pp. 25-34.

[6]. Михайлов В.И. Запоминающие устройства на оптических дисках / В.И. Михайлов, Г.И. Князев, П.П. Макарычев. -Москва: Радио и связь, 1991.-224 с. / Mikhailov V.I. Storage devices on optical disks/V.I. Mikhailov, G.I. Knyazev, P.P. Makarychev. -Moscow: Radio and Communications, 1991.-224 P.

[7]. Расчет электрических цепей и электромагнитных полей на ЭВМ/ Александрова М.Г., Белянин А.Н., Брукнер и др.: Под ред. Л. В. Данилов и Е. С. Филиппов. – Москва: Радио и связь, 1983. – 344 с. / Calculation of electrical circuits and electromagnetic fields on a computer/ M. G. Alexandrova, A. N. Belyanin, Bruckner et al.: Ed. L. V. Danilov and E. S. Filippov. – Moscow: Radio and communication, 1983. – 344 p.

[8]. Слесарев Ю. Н. Исследование аксиальной составляющей магнитного поля кольцевого магнита / Ю. Н. Слесарев, А.А. Воронцов//XX1 век: Итоги прошлого и проблемы настоящего. 2018 №4(44) Том 7, с. 92-96. / Slesarev Yu. N. Investigation of the axial component of the magnetic field of the ring magnet / Yu. N. Slesarev, AA Vorontsov //XX1 century: the results of the past and the problems of the present 2018 №4(44) Vol. 7, p. 92-96. /

[9]. Jen S. Magnetic and magneto-mechanical vibration properties of non-oriented electrical steel. / S. Jen, Y. Lin, C. Hsu, K. Lin // 2015 IEEE International Magnetics Conference (INTERMAG). – 2015. PP. 1 – 1.

[10]. N. Mohan, M. Sachdev, "A Static Power Reduction Technique for Ternary Content Addressable Memories," Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), 2004.

[11]. Martyshkin A I. Basic operation principles of associate coprocessor module for specialized computer systems based on programmable logical integral schemes. Journal of Fundamental and Applied Sciences, 2018, no. 10(6S), pp. 1449-1463.

[12]. Mathematical Modeling of Magnetic Fields of the Permanent Magnets and Solenoids, and Comparing the Results Obtained. Slesarev U.N., Mikhajlov P.G. and Vorontsov A.A. International Journal of Applied Engineering Research (IJAER) Volume 11, Number 20 (2016) pp. 10338–10342.

[13]. Vorontsov A.A., Slesarev Yu.N. Mathematical Modeling and Experimental Check of Output Signals of Magnetostrictive Converters of Movement. 2019 International Russian Automation Conference (RusAutoCon). – 2019. DOI 10.1109/RUSAUTOCON.2019.8867715.

[14]. Кулинг Х., Справочник по физике: Пер. с нем./ Под ред. Е. М. Лейкина. – М.: Мир, 1983. – 520 с./ Kuhling H., Handbook of physics: TRANS. with it./ Under the editorship of E. M. Leykin. – M.: Mir, 1983. – 520 P.

[15]. Бессонов Л. А. Теоретические основы электротехники. Электромагнитное поле: Учебник. – 9-е изд., перераб. и доп.. – М: Гардарики, 2001. – 528 с. / Bessonov L. A. Theoretical bases of electrical engineering. Electromagnetic field: Textbook. – 9th ed., – M: Gardariki, 2001. – 528 P.

[16]. Jia Yu. Fatigue Characteristics of Magnetostrictive Thin-Film Coated Surface Acoustic Wave Devices for Sensing Magnetic Field. / Y.Jia, W.Wang, Yu.Sun, M.Liu, X.Xue, Yu.Liang, Z.Du, J.Luo. // IEEE Access. - 2020. - Vol. 8. - P. 38347 - 38354. DOI 10.1109/ACCESS.2020.2976052.

[17]. Li Yu. High-Frequency Output Characteristics of Giant Magnetostrictive Transducer. / Yu.Li, W.Huang, B.Wang, L.Weng // IEEE Transactions on Magnetics. – 2019. - Vol. 55 (Issue 6). DOI 10.1109/TMAG.2019.2910854.

[18]. Купалян С. Д. Теоретические основы электротехники. Часть 3. Электромагнитное поле. Издание 3-е, исправленное и дополненное. – М.: Энергия, 1970. – 248 с./ Kupalyan S. D. Theoretical bases of electrical engineering. Part 3. Electromagnetic field. 3rd edition, revised and expanded. – M.: Energy, 1970. – 248 P.

[19]. Cozzolino J. Magnet engineering and test results of the high field magnet R&D program at BNL. / J. Cozzolino, M. Anerella, J. Escallier, G. Ganetis, A. Ghosh, R. Gupta, M. Harrison, A. Jain, A. Marone, J. Muratore, B. Parker, W. Sampson, R. Soika, P. Wanderer // IEEE Transactions on Applied Superconductivity. – 2003. - Vol.13, Issue 2, PP. 1347 – 1350.

[20]. Pradhan S. First Engineering Validation Results of SST-1 TF Magnets System. /S. Pradhan, K. Doshi, A. Sharma, U. Prasad, Y. Khristi, V. Tanna, Z. Khan, A. Varadharajalu, D. Sharma, M. Vora, A. Singh, B. Parghi, M. Banaudha, J. Dhongde, P. Varmora, D. Patel // IEEE Transactions on Applied Superconductivity. – 2014. - Vol. 24, Issue 3, Art. Seq. Num. 4301904.

Информация об авторах/ Information about authors

Артем Алексеевич ИВЗАНОВ – студент 4 курса по специальности 09.03.04 «Программная инженерия». Сфера научных интересов: магнитострикционные приборы, нейросети, искусственный интеллект.

Artem Alekseevich IVZANOV – 4th–year student with a degree in Software Engineering on 09.03.04. Research interests: magnetostrictive devices, neural networks, artificial intelligence.

Александр Анатольевич ВОРОНЦОВ – кандидат технических наук, доцент, доцент кафедры “Программирование” с 2012 года. Сфера научных интересов: магнитострикционные приборы, цифровая обработка сигналов, процессы и явления в магнитострикционных приборах.

Aleksander Anatolievich VORONTSOV – Candidate of Technical Sciences, Associate Professor, Associate Professor of the Department of Programming since 2012. Research interests: magnetostrictive devices, digital signal processing, processes and phenomena in magnetostrictive devices.

Юрий Николаевич СЛЕСАРЕВ – доктор технических наук, доцент, профессор кафедры «Автоматизация и управление» с 2012 года. Сфера научных интересов: магнитострикционные приборы, цифровая обработка сигналов, процессы и явления в магнитострикционных приборах, цифровая магнитная запись, энергетика, альтернативная энергетика.

Yuriy Nikolaevich SLESAREV – Doctor of Technical Sciences, Associate Professor, Professor of the Department of Automation and Control since 2012. Research interests: magnetostrictive devices, digital signal processing, processes and phenomena in magnetostrictive devices, digital magnetic recording, energy, alternative energy.

DOI: 10.15514/ISPRAS-2023-35(3)-4

Development and Implementation of the Digital Steganography Method Based on the Embedding of Pseudoinformation

I.G. Gvozdeva ORCID: 0009-0009-1058-2618 <*gvozdeva-irina@bk.ru*>
A.S. Gromov ORCID: 0009-0000-7130-6785 <*gromov.bo@yandex.ru*>
O.M. Gvozdeva ORCID: 0009-0000-1947-9638 <*olgagvozdevaaa@yandex.ru*>

Penza State University of Architecture and Construction,
28, Herman Titov st., Penza, 440028, Russia.

Abstract. The article provides an overview of the main methods of steganography, on the basis of which a new method was developed, consisting in embedding additional text (pseudo-information) in parallel with the transmitted message. An algorithm of this method has been developed. In this case, the frequency of the bit sequence was obtained in accordance with the generated pseudo-random numbers. In accordance with the algorithm, an application has been developed that allows the sender to encrypt and place the message in a container that is an image, and the recipient to determine the presence of the message and, if there is one, extract it. A computational experiment was also conducted, which showed that an image with a fairly large embedded text does not visually differ from the original image.

Keywords: steganography; cryptography; stegosystem; steganalysis.

For citation: Gvozdeva I.G., Gromov A.S., Gvozdeva O.M. Development and implementation of the digital steganography method based on the embedding of pseudoinformation. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 63-70. DOI: 10.15514/ISPRAS-2023-35(3)-4

Разработка и реализация метода цифровой стеганографии на основе встраивания псевдоинформации

И.Г. Гвоздева, ORCID: 0009-0009-1058-2618 <*gvozdeva-irina@bk.ru*>
А.С. Громов, ORCID: 0009-0000-7130-6785 <*gromov.bo@yandex.ru*>
О.М. Гвоздева, ORCID: 0009-0000-1947-9638 <*olgagvozdevaaa@yandex.ru*>

Пензенский государственный университет архитектуры и строительства,
440028, Россия, г. Пенза, ул. Германа. Титова, д. 28.

Аннотация. В статье приведен обзор основных методов стеганографии, на основании которого был разработан новый метод, заключающийся в встраивании дополнительного текста (псевдоинформации) параллельно с передаваемым сообщением. Разработан алгоритм этого метода. При этом частоту последовательности битов получали в соответствии с сгенерированными псевдослучайными числами. В соответствии с алгоритмом разработано приложение, позволяющее отправителю зашифровать и поместить сообщение в контейнер, представляющий собой изображение, а получателю определить наличие сообщения и, если оно имеется, извлечь его. Также был проведен вычислительный эксперимент, который показал, что изображение с довольно большим встроенным текстом визуально не отличается от исходного изображения.

Ключевые слова: стеганография; криптография; стегосистемы; стегоанализ.

Для цитирования: Гвоздева И.Г., Громов А.С., Гвоздева О.М. Разработка и реализация метода цифровой стеганографии на основе встраивания псевдоинформации. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 63–70 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–4

1. Introduction

The problem of delivering a confidential message has stood at all times. This problem has given rise to such sciences as cryptography and steganography.

The essence of steganography is that the message is placed in a container in such a way that an uninitiated circle of people sees only the object, not realizing that it may be filled with something. Here are some well-known examples: in ancient Greece, wooden writing boards covered with wax, under which there was a message, the heads of slaves with the message printed hidden under the hair, later, the so-called sympathetic ink, invisible under normal conditions, was widely used [1].

In the modern world, with the development of computer technology, digital data, as a rule, files of multimedia objects (images, video, audio, textures of 3D objects) serve as containers for hiding information. This is due to the fact that digitized objects, which initially have an analog nature, always have quantization noise, and when reproducing these objects, additional analog noise appears [2]. All this contributes to greater invisibility of hidden information.

The advantage of steganography over cryptography is that not only the contents of the transmitted message are hidden, but the very fact of the existence of this message is hidden.

In the science of steganography, such a direction as steganalysis is distinguished. The task of steganalysis is to identify the fact of transmission of hidden information in the analyzed message [3, 4]. Consequently, with the development of steganalysis, new methods are required to make hidden information inaccessible to the uninitiated [5]. In this article, the authors propose a way to embed information in a container that increases the reliability of its protection against unauthorized access.

2. Relevance

Digital steganography as a science was born literally in recent years. It includes the following areas:

- embedding information for the purpose of its hidden transmission;
- embedding of digital watermarks (CVZ) (watermarking);
- embedding identification numbers (fingerprinting);
- embedding titles (captioning).

This work touches on the first direction. Many methods and algorithms of steganography are known today. Here are some of them.

- LSB-steganography (the message is hidden in the lower bits (it is possible to use one or more lower bits) of the container [6, 7].
- The method based on hiding data in the coefficients of the discrete cosine transformation (hereinafter DCP) is a variation of the previous method, which is actively used, for example, when embedding a message in a JPEG format container.
- The method of hiding information using the lower bits of the palette – this method is essentially a variation of the general LSB method, but the information is embedded not in the least significant bits of the container, but in the least significant bits of the palette. As a result, the container capacity is low.
- The method of hiding information in the service fields of the format is a method in which the embedded message is placed in the service fields of the container header. Obvious disadvantages are the low capacity of the container and the ability to detect embedded data using conventional image viewing programs (which sometimes allow you to see the contents of service fields).

As early as 1883, Kergoff wrote that the information security system should provide its functions even with full awareness of the enemy about its structure and algorithms of functioning.

This means that the message embedding model should be sufficiently complex so that the enemy, if he guesses about the presence of a hidden message, even with the presence of powerful computing equipment, would not be able to extract it [8].

In connection with the above, the authors propose a method of concealing information, which aims to increase confidence that the message intercepted by the enemy will not be opened.

This method is based on the fact that false, so-called pseudo-information is embedded in parallel with useful information. When selecting a model for extracting text from a container file, the opponent relies on the result obtained, which represents any characters. And it is not possible to distinguish the symbols belonging to useful information from false information.

3. Algorithm for embedding information in a container and extracting it

Our proposed method is based on the LSB method, the message will be hidden in the lower bits of the image. A broadband method was used to select the sequence of bits. Such transmission methods are used in communication technology to ensure high noise immunity and complicate the interception process. The purpose of broadband methods is similar to the tasks that a stegosystem solves: to try to “dissolve” a secret message in a container and make it impossible to detect it. Since signals distributed over the entire spectrum band are difficult to detect, steganographic methods based on broadband methods are resistant to accidental and intentional distortion. In this work, the method of jumping frequencies was used, when the frequency of bits intended for embedding information changes according to some pseudo-random law. The frequency of using a byte of color is also randomly selected.

Similarly, frequencies are generated for embedding pseudo-information that do not intersect with the received numbers to accommodate the basic information. The generated frequencies are stored and must be transmitted to the receiving party and are a cryptographic key.

A bmp graphic file with an RGB palette model with a coding depth of 24 bits (8 bits per color) was taken as a container file.

The contents of the container file and the file to be hidden are placed in byte arrays.

For embedding, two random of the four lower bits of one of the three components of the color are used. Since 3 bytes form one color, one byte of text will have 12 bytes of graphics. Before starting the implementation of the algorithm, you should check whether the text file fits into the graphic. The following is the embedding algorithm.

- The last two bits in the specified color component are "released". To do this, the corresponding byte of color is multiplied by a byte mask, with zeros in the specified bits using bitwise multiplication. As a result, these two bits will be reset to zero.
- Take the first two bits from the byte-"text". To do this, multiply the byte "text" by the byte mask equal to 192 (11000000).
- In the resulting byte, we will shift to the right. As a result, the first two bits will be in the specified two places.
- Add the received byte to the edited byte obtained in the first paragraph using bitwise addition. As a result, the first two bits of the text are "hidden".
- Further actions will be repeated.
- After reading the next byte of text, and the actions starting from point 1 are repeated.
- The size of the text is recorded in one of the free bytes of the header part of the graphic file.

Similar actions are performed for embedding pseudo-information. When extracting a message from an image, the reverse action is performed in accordance with the available encryption key.

To test the algorithm's operability, an application was developed that allows the sender to hide the message and the recipient to extract it.

4. Requirements for an application that implements the part of the stegosystem in which information is embedded and extracted

We proceed to the formulation of the requirements for the application. The user should be able to perform the following functions:

- selecting a file with a message to embed;
- selecting a text file for embedding pseudo-information;
- selecting an image file for the shorthand algorithm;
- selecting the name of the resulting image file that will contain encrypted data;
- selecting an image file containing encrypted data;
- selecting the name of the resulting file that will contain the extracted data.

Let's show the user's interaction with the application on the use case diagram (*Fig. 1*).

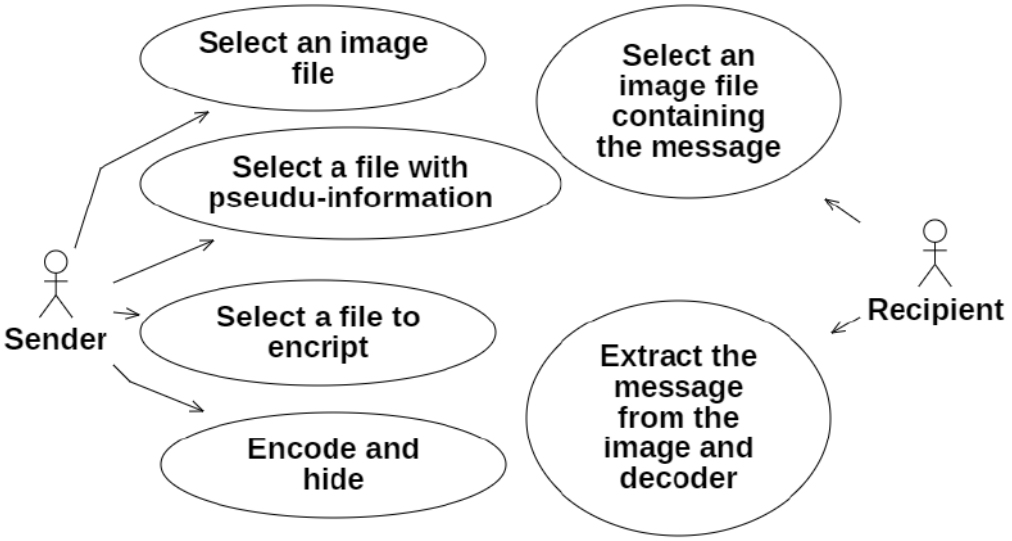


Fig. 1. User interaction with the application

In the implemented system, the text to be embedded is pre-encoded by the byte permutation method, and then by the bit permutation method in accordance with the pseudo-random sequence. Let's describe the logic of the system behavior using the diagram shown in the *Fig. 2*.

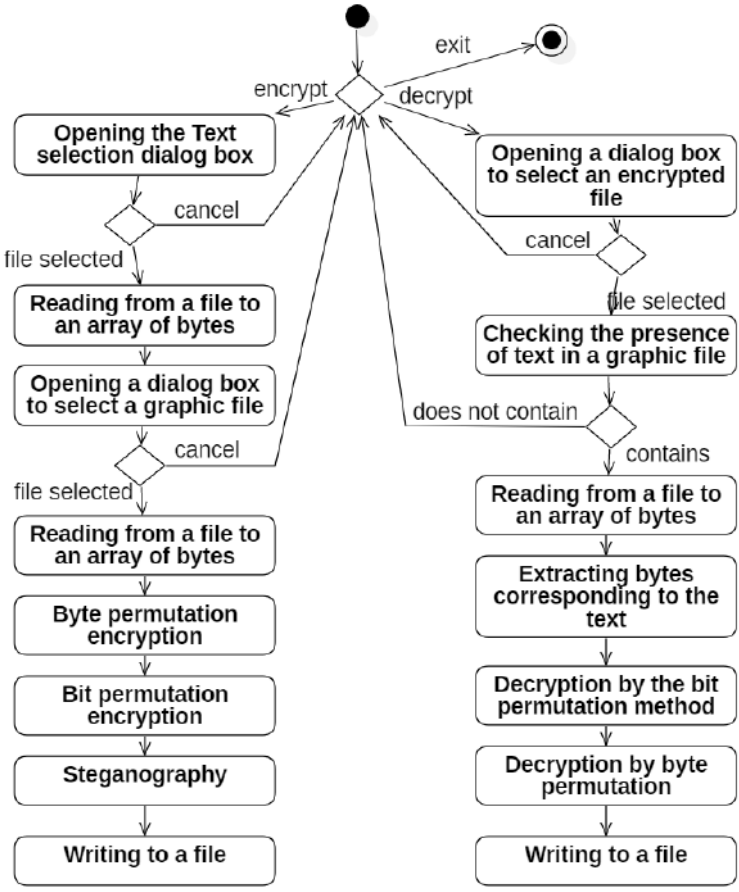


Fig. 2. State diagram

5. Description of the program and test results

In accordance with the designed diagrams, an application was developed.

When the application is launched, the main window appears, providing the user with two functions: encrypt information or decrypt (*Fig. 3*).

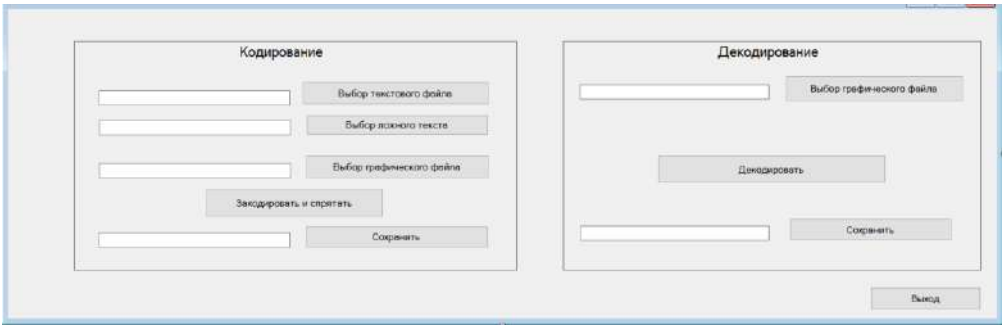


Fig. 3. Main application window

When you click the Select Text File button, a file selection window appears (*Fig. 4*).

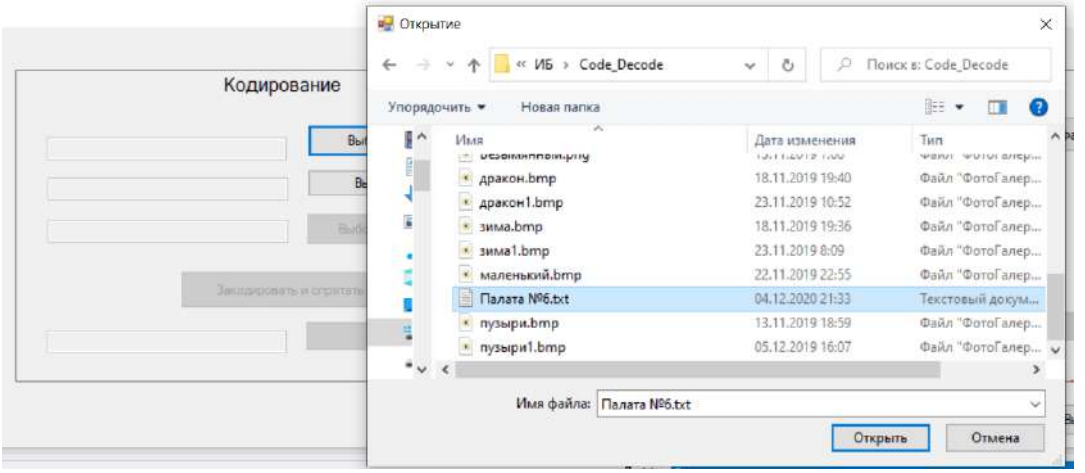


Fig. 4. Selecting a text file

After that, the Select Image File button becomes available. When you click on it, a similar window will open. If the selected image file is too small to contain text information, the file will not open and a message will be displayed about it (*Fig. 5*).

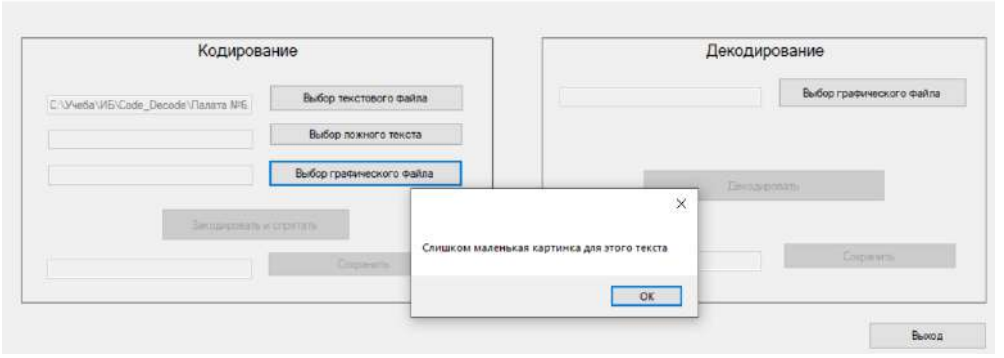


Fig. 5. Insufficient file size

If everything was successful, the Encode and Hide button will be available. When you click on it, the byte and bit permutation methods are applied sequentially, then the steganography method works and the Save button becomes available.

When you select the Select Image File button, it checks whether the file contains hidden text. If not, the file does not open and a message about it is displayed (*Fig. 6*)

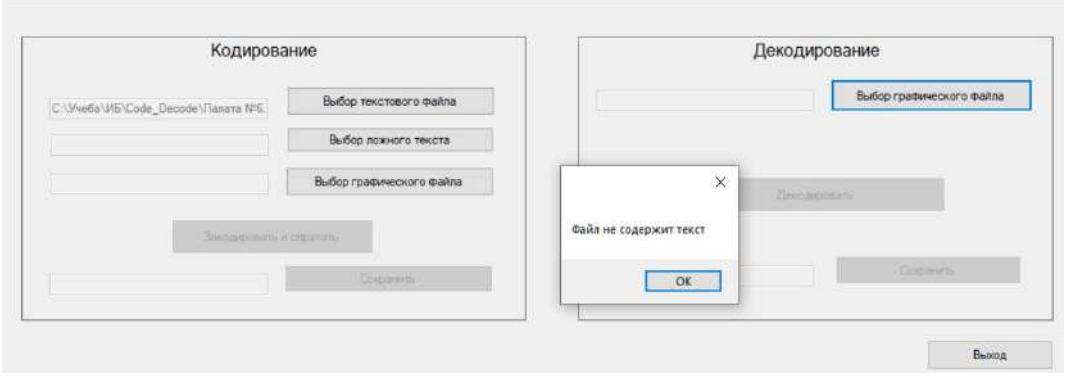


Fig. 6. Opening a file that does not contain text

Further actions are similar to those described above.

Testing was carried out using graphic files "bubbles.bmp" size 2344 KB, "small.bmp" with a size of 1 KB and a text file "Chamber No.6.txt " size 103 KB. The result of the hidden file was recorded in the file "bubbles 1.bmp".

Fig. 7 shows the original graphic file and the file with hidden text.

Fig. 7. Image files: with and without embedded text

As can be seen, the replacement of two bits in one color component did not produce any visible changes, which allows using this method along with existing stegosystems.

Conclusion

Thus, as a result of the review of existing methods of steganography, a new method of hiding messages was proposed and implemented, in which the use of embedding pseudo-information is proposed. The results of the development are presented for digital images of the BMP format, however, they can be adapted to other formats.

References

[1]. Gribunin V.G., Okov. I. N. and Turintsev I. V., “Digital steganography” [Text],. Moscow : SOLON-Press, 263 p., 2003./ Грибунин В. Г., Оков И. Н., Туринцев И. В.. Цифровая стеганография [Текст] – Москва: СОЛОН-Пресс, 2003. – 263 с.

[2]. Razinkov E. V., Latypov R. H. “Stability of steganographic systems”, Scientific notes of Kazan.state University, Kazan, Vol. 151, No 2, 2009./ Разинков Е. В., Латыпов Р. Х. Стойкость стеганографических систем // Ученые записки Казанского государственного университета. Сер.: Физико-математические науки. 2009. Т. 151, кн. 2. С.126-132.

[3]. Golubev E. A., Varnovsky N. P. and Logachev O. A., Conference "Mathematics and Security of Information Technologies", Moscow State University, Moscow, Russia, October 2004, 28-29./ Голубев Е. А., Варновский Н. П. и Логачев О. А., Конференция "Математика и безопасность информационных технологий", Московский государственный университет, Москва, Россия, октябрь 2004, 28-29.

[4]. Fridrich J., Du R., Long M. “Steganalysis of LSB encoding in color images”, ICME, 2000.

[5]. Replacement of the least significant bit [Electronic resource]. – Access mode: http://www.nestego.ru/2012/07/lsb.html .

[6]. Provos N., Honeyman P. “Detecting Steganographic Content on the Internet” // Proceeding of the 10 USENIX Security Symposium,. pp. 323–335, 2001.

[7]. Westfeld A. “Attacks on Steganographic Systems: Breaking the Steganographic Utilities EzStego, Jsteg, Steganos and S-Tools-and Some Lessons Learned “/ A. Westfeld, A. Pfitzmann // 3rd International Workshop on Information Hiding (2000)

[8]. Zakalkin P. V., Ivanov S. A., Vershennik E. V. and Kiryanov A. V., “Method of masking transmitted information”, Proceedings of ISP RAS, 32:6 (2020), pp 111–126./

Закалкин П.В., Иванов С.А., Вершенник Е.В., Кирьянов А.В. Способ маскирования передаваемой информации. Труды ИСП РАН, том 32, вып. 6, 2020 г., стр. 111-126.

Информация об авторах / Information about authors

Ирина Геннадьевна ГВОЗДЕВА – кандидат технических наук, специалист кафедры Информационно-вычислительных технологий Пензенского государственного университета архитектуры и строительства. Сфера научных интересов: область математического моделирования и оптимального управления технологическими процессами в строительстве, электрохимии, экологии.

Irina Gennadievna GVOZDEVA – Candidate of Technical Sciences, specialist of the Department of Information and Computing Technologies of the Penza State University of Architecture and Construction. Research interests: mathematical modeling and optimal control of technological processes in construction, electrochemistry, ecology.

Артем Сергеевич ГРОМОВ – студент, обучающийся по специальности «Информационные системы и технологии» в Пензенском государственном университете архитектуры и строительства. Область научных интересов: защита информации.

Artem Sergeyevich GROMOV is a student studying in the specialty "Information Systems and Technologies" at the Penza State University of Architecture and Construction. Research interests: information protection.

Ольга Михайловна ГВОЗДЕВА – студент, обучающийся по специальности «Информационные системы и технологии» в Пензенском государственном университете архитектуры и строительства. Область научных интересов: управление данными.

Olga Mikhailovna GVOZDEVA is a student studying in the specialty "Information Systems and Technologies" at the Penza State University of Architecture and Construction. Research interests: data management.

DOI: 10.15514/ISPRAS-2023-35(3)-5

Deployment Approaches in Distributed Complex Event Processing

A.A. Zorin, ORCID: 0009-0000-2689-2543 <zorinarsenij@mail.ru>
I.E. Chernetskaya, ORCID: 0009-0009-8254-9606 *<white731@yandex.ru>*
Southwest State University,
94, ul. 50 Let Oktyabrya, Kursk, Russia, 305040.

Abstract. Big Data technologies have traditionally focused on processing human-generated data, while neglecting the vast amounts of data generated by Machine-to-Machine (M2M) interactions and Internet-of-Things (IoT) platforms. These interactions generate real-time data streams that are highly structured, often in the form of a series of event occurrences. In this paper, we aim to provide a comprehensive overview of the main research issues in Complex Event Processing (CEP) techniques, with a special focus on optimizing the distribution of event handlers between working nodes. We introduce and compare different deployment strategies for CEP event handlers. These strategies define how the event handlers are distributed over different working nodes. In this paper we consider the distributed approach, because it ensures, that the event handlers are scalable, fault-tolerant, and can handle large volumes of data.

Keywords: complex event processing; distributed processing; event based systems.

For citation: Zorin A.A., Chernetskaya I.E. Deployment approaches in distributed complex event processing. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 71-82. DOI: 10.15514/ISPRAS-2023-35(3)-5

Подходы к развертыванию в распределенной обработке сложных событий

А.А. Зорин, ORCID: 0009-0000-2689-2543 *<zorinarsenij@mail.ru>*
И.Е. Чернецкая, ORCID: 0009-0009-8254-9606 *<white731@yandex.ru>*
Юго-Западный государственный университет,
50 лет Октября ул., 94, Курск, Курская обл., 305040.

Аннотация. Технологии больших данных традиционно фокусировались на обработке данных, генерируемых человеком, пренебрегая при этом огромными объемами данных, генерируемых межмашинными взаимодействиями и платформами Интернета вещей. Эти взаимодействия генерируют потоки данных в реальном времени, которые являются высокоструктурированными, часто в виде серии событий. В этой статье мы стремимся предоставить всесторонний обзор основных исследовательских проблем в области методов комплексной обработки событий (CEP), уделяя особое внимание оптимизации распределения обработчиков событий между рабочими узлами. Мы представляем и сравниваем различные стратегии развертывания обработчиков событий CEP. Эти стратегии определяют, как обработчики событий распределяются по различным рабочим узлам. В этой статье мы рассматриваем распределенный подход, поскольку он гарантирует, что обработчики событий масштабируемы, отказоустойчивы и могут обрабатывать большие объемы данных.

Ключевые слова: обработка сложных событий; распределенная обработка; системы, основанные на событиях.

Для цитирования: Зорин А.А., Чернецкая И.Е. Подходы к развертыванию в распределенной обработке сложных событий. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 71–82 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–5

1. Introduction

Several complex systems operate by observing a set of primitive events that happen in the external environment, interpreting and combining them to identify higher level composite events, and finally sending notifications about these events to the components in charge of reacting to them, thus determining the overall system's behavior. This means that the systems are able to perform complex tasks by breaking them down into simpler, more manageable events. In order to achieve this, the systems use a general architecture that includes sources and sinks at the peripherals of the system. These sources observe primitive events and report them, while the sinks receive composite event notifications and react to them.

At the center of the system is the complex event processing (CEP) subsystem, which is responsible for processing and routing events from sources to interested sinks. It operates by interpreting a set of event definition rules, which describe how composite events are defined from primitive ones [1, 2]. The CEP subsystem is crucial to the operation of the system, as it is responsible for ensuring that the right events get to the right places.

Event-based applications usually involve a large number of sources and sinks, possibly dispersed over a wide number of working nodes [3, 4, 5]. This means that the CEP subsystem can be internally built around several, distributed working nodes, connected together to form an overlay network, and cooperating to provide the processing and routing service [6]. This allows the system to process and route events more efficiently, as it can distribute the workload across multiple working nodes.

This paper introduces and compares different deployment approaches for CEP, which are designed to optimize the performance of the system. A deployment approach defines how the event handlers are distributed over working nodes. The first aspect is often called operator placement, and it involves finding the best mapping of the event handlers defined in rules on available working nodes [7]. Operator placement may pursue different goals, such as reducing the latency required to deliver notifications to interested parties, or minimizing the usage of network resources. In the last few years, different solutions have been proposed for operator placement. However, the problem is known to be extremely complex to solve, even for small instances with a reduced number of workers and rules. Accordingly, existing approaches are often based on approximated optimization algorithms or heuristics, and they usually rely on a centralized decider, which collects all the relevant information about the network status and locally computes a solution to the problem.

The novelty of this work is the study of the applied use of scaling approaches in systems for processing complex events in real time.The solutions presented in this paper are explicitly tailored to large scale distributed scenarios. They try to take into account the topology of the processing network as well as the location of event sources and their generation rates [8].

2. Approaches

2.1. Uniform distribution of handlers between working nodes

This approach for distributing handlers is based on an even distribution of handlers among all the working nodes. The implementation of this approach is simple and requires a few steps. Firstly, the handler distribution storage must be expanded to include information about the number of running handlers on each of the working nodes. The data schema in DBML format might look like this:

```
Table handlers {
  id integer [primary key]
  w_node_id integer
  other_data data
}


Table working_nodes {
  id integer [primary key]
  other_data data
}

Ref: working_nodes.id > handlers.w_node_id
```

The volume of the information storage depends on the number of working nodes and handlers, but does not depend on the number of events occurring in the system. Therefore, the memory cost for storing the information can be estimated in O(W+H), where W is the number of working nodes, and H is the number of handlers.

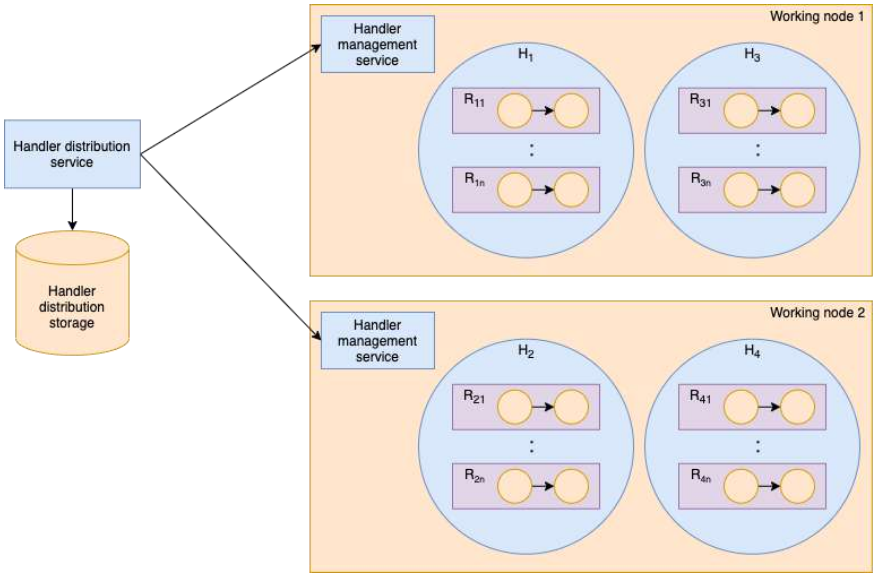


Fig. 1. Uniform distribution of handlers between working nodes

Once this information is available, the handler distribution service can be used to control the even launch of handlers across all working nodes. Fig. 1 illustrates this approach with the uniform distribution of four handlers between two working nodes. The handler distribution storage is used to store information about the handlers that are running on specific working nodes and their numbers. If there is a change in the number of handlers, the handler distribution service will redistribute them. When a new handler is added, the handler distribution service identifies the working node with the fewest running handlers and deploys the new handler to that node. Conversely, when a handler is removed, the handler distribution service removes information about the handler from the handler distribution storage and sends a handler shutdown command to the handler management service. However, removing handlers may cause an imbalance in the number of handlers on each working node.

To solve this issue, the handler distribution service periodically balances the number of handlers on each working node. The service first determines the maximum number of handlers allowed on each working node using the following formula:

$$N = \left\lceil \frac{H}{W} \right\rceil$$

In (1) H is the number of event handlers and W – the number of working nodes. It then sequentially traverses the sorted list of working nodes, and if the number of running handlers on the working node is more than the maximum number allowed, the service searches for working nodes with a number of handlers less than the maximum allowed. The excess handlers from the current working node are transferred to the new working nodes. The handlers redistribution algorithm will look like this:

Algorithm 1 Function RedistributeHandlers(W,H)

```
1: w_start ← 0
2: w_end ← len(W) − 1
3: n ← len(H)/len(W)
4: for w_start ≤ w_end do
5:    if W[w_start].number_of_handlers < n then
6:       for w_start ≤ w_end do
7:          if W[w_end].number_of_handlers > n then
8:             Redistribute(W[w_start], W[w_end])
9:             if W[w_start].number_of_handlers ≥ n then
10:               break
11:            end if
12:         end if
13:         w_end ← w_end − 1
14:      end for
15:   end if
16:   w_start ← w_start + 1
17: end for
```

The asymptotic complexity of the algorithm in such an implementation is equal to O(max(W,H)). Although this approach is easy to implement and allows for horizontal scaling of handlers, it has some inherent disadvantages. For instance, it does not take into account the internal complexity of each handler or possible differences in the number of resources on the working node. Each handler may contain a different number of rules, and the frequency of rule triggering may vary. Additionally, working nodes may have differing amounts of resources, which can lead to low efficiency in the distribution of handlers across working nodes.

2.2. Distribution of handlers based on the number of rules

This approach shares similarities with the previous one, but there is a key difference in how the handlers are distributed. Instead of relying on a simple criterion, such as the number of active handlers, this approach takes into account the number of handlers running on each working node. To accomplish this, the handler distribution storage is expanded to include information about the number of rules in each handler. The extended data schema in DBML format for that approach might look like this:

```
Table handlers {
  id integer [primary key]
  number_of_rules integer
  w_node_id integer
```

```
  other_data data
}

Table working_nodes {
  id integer [primary key]
  other_data data
}

Ref: working_nodes.id > handlers.w_node_id
```

This allows for a more nuanced approach to balancing the workload between working nodes, which is illustrated on fig. 2.

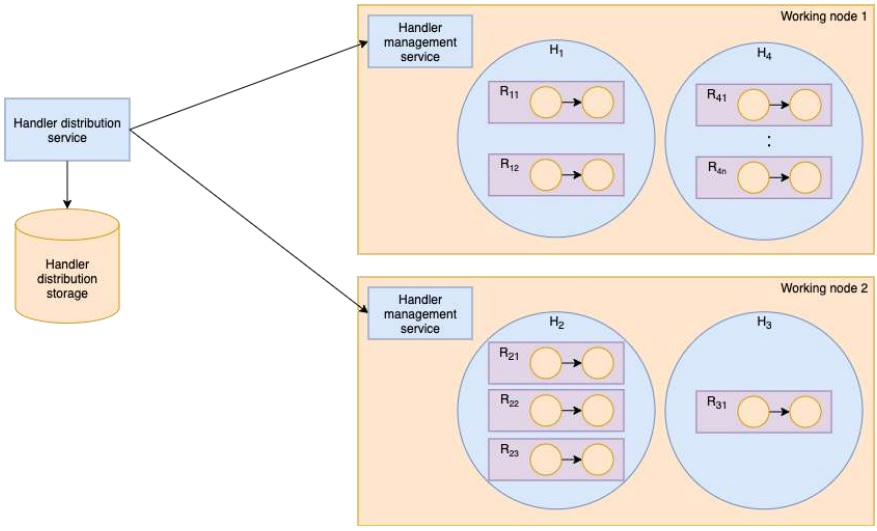


Fig. 2. Distribution of handlers based on the number of rules

The volume of the information storage depends on the number of working nodes and handlers as for the previous approach. Therefore, the memory cost for storing the information can be estimated in O(W+H). The redistribution algorithm requires an analysis of the number of rules executed on the working node, instead of calculating the number of handlers. The complexity of the algorithm corresponds to the complexity of the previous algorithm and is equal to O(max(W,H)).

One of the main advantages of this approach is that the handler distribution service can monitor the total number of handler rules running on each working node. Like the previous approach, the handler distribution service performs balancing at fixed intervals. However, the key difference is the inclusion of additional information about the number of rules, which allows for a more complex balancing algorithm to be used. By evenly distributing handlers, this approach minimizes the number of rules executed on each working node, which can lead to more efficient processing. However, it's important to note that this approach still does not take into account the frequency of rule firing or the different amounts of available resources on working nodes, which could impact overall performance. Therefore, it may be necessary to explore additional strategies for optimizing the workload distribution in the future.

2.3. Distribution of handlers based on the configuration of the required resources

This approach involves a preliminary configuration of the necessary resources for each handler. The system administrator adds information about the resources that are needed for each handler and also

adds information about the resources available on each working node. With the help of this information, the distribution of handlers between working nodes takes place. The distribution process ensures that the resources of working nodes are utilized as much as possible. Before launching a network of handlers, the configuration of the resources required by each handler and the resources available on each working node is performed. The configurable resources can be the number of CPU cores and the size of RAM. In addition to being able to configure resources, this approach also allows for consideration of the frequency of execution of the rules by each handler. This frequency data could be used to optimize the distribution of handlers.

The volume of the information storage depends on the number of working nodes and handlers as for the previous approach. Therefore, the memory cost for storing the information can be estimated in O(W+H). The extended data schema in DBML format for that approach might look like this:

```
Table handlers {
  id integer [primary key]
  cpu_required integer
  memory_required integer
  w_node_id integer
  other_data data
}

Table working_nodes {
  id integer [primary key]
  cpu integer
  memory integer
  other_data data
}

Ref: working_nodes.id > handlers.w_node_id
```

The task of efficiently placing handlers in this approach is an NP challenge. Therefore, a resource allocation approach from kubernetes can be used to provide a trade-off between speed and efficiency [9]. In this case, the algorithm is reduced to calculating the estimate of the deployment of the handler on each of the working nodes [10]. The algorithmic complexity of this algorithm is O(W * H).

The scheme of this approach is shown in fig. 3.

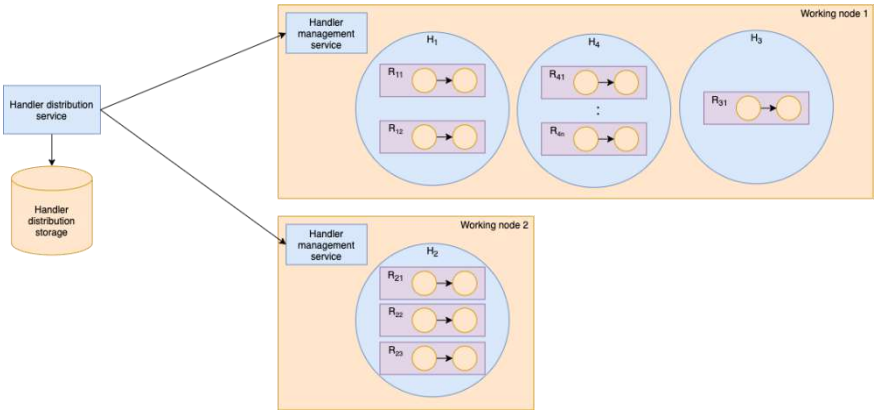


Fig. 3. Distribution of handlers based on the configuration of the required resources

However, one disadvantage of this approach is the need for manual configuration of allocated resources, which can be time-consuming. Another disadvantage is that this approach does not take

into account the dynamic nature of resource availability, which could lead to suboptimal resource utilization. To address these limitations, future research could explore the use of machine learning algorithms to automate the allocation of resources and dynamically adjust to changes in resource availability.

2.4. Distribution of handlers based on statistics collected during operation

All previous diagrams are based on information obtained from starting the entire system and creating new handlers. However, it is not always possible to determine how many resources to allocate to a handler and on which working node it is most efficient to place them. This problem is due to the fact that at the time the handlers are launched, there is no information about the frequency of the rule's operation. It is important to consider the frequency of rule execution when allocating resources because it can affect the efficiency of the handler. A handler may contain a large number of rules, but these rules are fired quite rarely [11]. In contrast, a handler may contain only one rule, but fire on most events. These scenarios can lead to resource waste or inefficient allocation. One way to solve this problem is to collect analytics from handlers while the system is running. Collecting statistics on the execution time and frequency of rules can help in balancing handlers with infrequently executed rules on less productive working nodes and those with the longest rule execution time and high execution frequency on high-performance working nodes. To collect statistics, it is most efficient to run the statistics storage locally on each working node. This will ensure the shortest time to send statistics from the handler to the statistics storage. Each handler sends all necessary statistics to the local statistics storage on the working node. The handler distribution service collects handler statistics from each working node through the handler management service during balancing. After that, the service aggregates the collected statistics and, based on the results, redistributes highly loaded processors to the most high-performance working nodes. This ensures that the system is balanced and optimized for efficient execution. The extended data schema in DBML format for that approach might look like this:

```
Table rules {
   id integer [primary key]
   processing_time_q95 integer
   number_of_activations integer
   h_id integer
}

Table handlers {
  id integer [primary key]
  w_node_id integer
  other_data data
}

Table working_nodes {
  id integer [primary key]
  cpu integer
  memory integer
  other_data data
}

Ref: working_nodes.id > handlers.w_node_id
Ref: handlers.id > rules.h_id
```

The volume of the information storage depends on the number of working nodes, handlers and rules. Therefore, the memory cost for storing the information can be estimated in O(W+H+R), where W is the number of working nodes, H is the number of handlers and R is the number of rules. Also, this approach uses local storage for rule execution statistic. This collected statistic can be collapsed, so the space used does not exceed O(R), since all statistics are duplicated in the handler distribution storage.

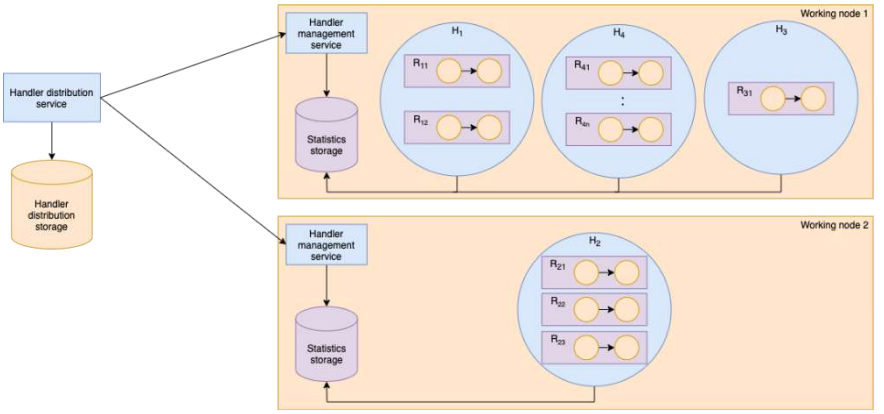


Fig. 4. Distribution of handlers based on statistics collected during operation

On fig. 4, we can see the distribution of handlers based on the statistics collected during the work. The diagram shows that each working node has local statistics storage. The handler distribution service, at the time of balancing, collects and aggregates data from local statistics storages and creates it. So, as shown in fig. 4, the handler distribution service receives information about the 95^{th} percentile of the rule execution time and the number of rule firings. Based on the aggregated statistics, the handler distribution service performs balancing and places the most loaded H_2 handler on a separate working node 2. This algorithm also reduces to solving the bin packing problem, like the previous one, and has a similar complexity - O(W * H).

In conclusion, collecting analytics can help in efficient resource allocation and balancing of handlers, leading to a more optimized system. By running the statistics storage locally on each worker node, the system can ensure the shortest time to send statistics from the handler to the statistics storage.

3. Comparison of approaches

Let's make a comparative analysis of the described schemes for working with events according to the following criteria [12]:

- Support for working with working nodes with different amounts of resources;
- Level of support for accounting for the frequency of operation of handler rules;
- The need to develop additional services and repositories with information storages;
- The complexity of the algorithm for redistributing handlers between working nodes.

Consider the rating scale for each criterion. The criterion for supporting work with working nodes with different amounts of resources can be evaluated on the following scale:

- Present - 1;
- Absent - 0.

The criteria for the level of support for accounting for the frequency of triggering of handler rules can be assessed on a scale:

- Dynamic support - 1;
- Static support - 0.5;

- Absent - 0.

Dynamic support implies the ability of the system to independently collect statistics on the frequency of rule triggering and, based on the collected data, balance handlers. Static support allows configuration of the frequency of rule triggering at the system startup stage. This approach does not allow efficient utilization of resources in the case of a changing frequency of rule firings over time.

The criteria for the need to develop additional services and repositories can be estimated based on the assessment of overhead costs for information storage. Thus, the criterion can be assessed on the following scale:

- Development of additional services and repositories is not required, no overhead - 1;
- Requires the development of information storage, the volume of which does not depend on the number of rules specified - 0.5;
- Requires the development of information storage, the volume of which depends on the number of rules or a value of a higher order - 0.

The criteria for the complexity of the algorithm for redistributing handlers between working nodes can be estimated using the following scale:

- Algorithm complexity not exceeding O(max(W,H)) - 1;
- Algorithm complexity not exceeding O(W * H) - 0.5;
- Algorithm has quadratic complexity and higher - 0.

Criteria 1 and 2 are the most important as they affect the efficiency of resource utilization at working nodes [13]. Therefore, the weight of criteria 1 and 2 is 0.3, and the weight of criterion 3 and 4 is 0.2. The weighted sum method shows (Tab. 1) that the approach of distributing handlers based on run time statistics is more appropriate.

Table 1. Comparison by weighted sum method

Criteria	Approaches			
	A	B	C	D
C_1	0	0	1	1
C_2	0	0	0.5	1
C_3	0.5	0.5	0.5	0
C_4	1	1	0.5	0.5
Weighted sum	0.35	0.35	0.65	0.7

It allows working with working nodes that have different amounts of resources and provides a redistribution of handlers between working nodes, taking into account the actual frequency of rule firing. This approach also has disadvantages in the form of the need to create additional local storage of statistics and implement the aggregation of the collected statistics.

4. Conclusion and future work

Having thoroughly reviewed the state-of-the-art approaches that focus on efficient event handler distribution and can be applied in CEP systems. We have come to the conclusion that the approach

using statistics collected during the operation of the system to redistribute handlers between working nodes is the most suitable approach for modern systems. This approach utilizes not only the static configuration of the distribution strategy at the stage of system startup but also dynamic redistribution based on statistics collected during the operation of the system. This can improve the efficiency of resource utilization in the system. Therefore, we recommend that future research focus on the study of hybrid approaches to managing the distribution of handlers between working nodes, where both static configuration and dynamic redistribution can be used to maximize system efficiency.

In addition to this, we suggest that it would be beneficial to select the optimal set of metrics that can effectively redistribute event handlers. Further research in this area may lead to the identification of the most relevant metrics.

Although we have considered centralized approaches to managing the distribution of event handlers in this work. There are also decentralized approaches that provide a higher level of fault tolerance and have the potential to scale efficiently [14,15]. Therefore, we suggest that future work may explore these decentralized approaches as well. By investigating both centralized and decentralized approaches, we can gain a better understanding of the advantages and disadvantages of each and ultimately identify the best approach for a given system.

References

[1]. Paschke A., Kozlenkov A. Rule-Based Event Processing and Reaction Rules: Lecture Notes in Computer Science, 2009, pp. 53-66.
[2]. Cugola G., Margara A. Deployment strategies for distributed complex event processing: Computing, 2012, vol. 95, no. 2, pp. 129-156.
[3]. Fardbastani M., Sharifi M. Scalable complex event processing using adaptive load balancing: Journal of Systems and Software, 2019, v. 149, pp. 305-317.
[4]. Sun A., Zhong Z., Jeong H., Yang Q. Building complex event processing capability for intelligent environmental monitoring: Environmental Modelling and Software, 2019, v. 116, pp. 1-6.
[5]. Loreti D., Chesani F., Mello P., Roffia L., Antoniazzi F., Cinotti T., Paolini G., Masotti D., Costanzo A. Complex reactive event processing for assisted living: The Habitat project case study: Expert Systems with Applications, 2019, v. 126, pp. 200-217.
[6]. Brazález E., Macià H., Díaz G., Baeza Romero M., Valero E., Valero V. FUME: An air quality decision support system for cities based on CEP technology and fuzzy logic: Applied Soft Computing, 2022, v. 129, pp. 109536.
[7]. Paschke A., Kozlenkov A., Rule-Based Event Processing and Reaction Rules: Lecture Notes in Computer Science, 2009, pp. 53-66.
[8]. Alakari A., Li K. F., Gebali F., A situation refinement model for complex event processing, Knowledge-Based Systems [online] 198, 2020, 105881.
[9]. Hightower K., Burns B., and Beda J., Kubernetes: Up and Running: Dive into the Future of Infrastructure, O'Reilly Media, 2017.
[10]. Luksa M., Kubernetes in Action, Hanser Fachbuchverlag, 2018, ISBN 9783446455108.
[11]. Wang D., Zhou M., Ali S., Zhou P., Liu Y., Wang X., A Novel Complex Event Processing Engine for Intelligent Data Analysis in Integrated Information Systems: International Journal of Distributed Sensor Networks, 2016, vol. 12, no. 3, pp. 6741401.
[12]. Alakari A., Li K. F., Gebali F., A situation refinement model for complex event processing, Knowledge-Based Systems [online] 198, 2020, 105881.
[13]. Margara A., Cugola G., High-Performance Publish-Subscribe Matching Using Parallel Hardware: IEEE Transactions on Parallel and Distributed Systems, 2014, vol. 25, no. 1, pp. 126-135.
[14]. Cugola G., Margara A., Complex event processing with T-REX: Journal of Systems and Software, 2012, vol. 85, no. 8, pp. 1709-1728.
[15]. Jayasekara S., Kannangara S., Dahanayakage T., Ranawaka I., Perera S., Nanayakkara V., Wihidum: Distributed complex event processing: Journal of Parallel and Distributed Computing, 2015, vol. 79-80, pp. 42-51.

Информация об авторах / Information about authors

Арсений Андреевич ЗОРИН – аспирант кафедры вычислительной техники Юго-Западного государственного университета. Его научные интересы включают обработку сложных событий в распределенных системах, интеллектуальные системы на основе правил.

Arsenij Andreevich ZORIN is a post–graduate student of the Department of Computer Engineering of Southwest State University. His research interests include processing complex events in distributed systems, intelligent rule-based systems.

Ирина Евгеньевна ЧЕРНЕЦКАЯ - заведующий кафедрой вычислительной техники Юго-Западного государственного университета, доктор технических наук. Её научные интересы включают математическое и алгоритмическое описание сложных технологических процессов, разработка автоматизированных систем управления.

Irina Evgenyevna CHERNETSKAYA - Head of the Department of Computer Engineering of Southwestern State University, Doctor of Technical Sciences. Her research interests include mathematical and algorithmic description of complex technological processes, development of automated control systems

DOI: 10.15514/ISPRAS-2023-35(3)-6

Mathematical Modeling and Software for Calculating Regimes of Galvanic Wastewater Purification from Heavy and Nonferrous Metals in Devices with Flow-Through Three-Dimensional Electrodes

[1] *V.V. Kuzina*, ORCID: 0000-0003-4511-7176 <*kuzina@pguas.ru*>
[2] *V.K. Varentsov*, ORCID: 0000-0001-8622-9364 <*vvk@ngs.ru*>
[1] *A.N. Koshev*, ORCID: 0000-0003-3057-4980 <*koshev@pguas.ru*>
[1] *G.M. Kupriyanko*, ORCID: 0009-0001-6737-554X <*gtaas@mail.ru*>

[1] *Penza State University of Architecture and Construction, 28, German Titov st., Penza, 440028, Russia.*
[2] *Institute of Solid State Chemistry and Mechanochemistry of Siberian branch of Russian Academy of Science, 18, Kutateladze st., Novosibirsk, 630128, Russia.*

Abstract. The paper briefly discusses progressive technologies of wastewater treatment from ions of heavy and nonferrous metals of industrial and small enterprises of urban agglomerations. An analysis of the efficiency of three-dimensional flow-through electrodes for wastewater treatment of harmful reagents is given. The mathematical models of electrochemical processes in three-dimensional flow-through electrodes as applied to extract metals from the solutions of galvanochemical industries are presented. The description of the set of programs developed in the programming language Object Pascal for computational experiments according to the obtained mathematical models is given. Numerical solution of scientific problem of practical importance has been obtained by using the program complex. A good correspondence between the results of calculations and experiments is shown.

Keywords: mathematical modeling; software package; Object Pascal programming language; three-dimensional flow electrode; extraction of metals from galvanochemical solutions.

For citation: Kuzina V.V., Varentsov V.K., Koshev A.N., Kupriyanko G. Mathematical modeling and software for calculating regimes of galvanic wastewater purification from heavy and nonferrous metals in devices with flow-through three-dimensional electrodes. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 83-90. DOI: 10.15514/ISPRAS-2023-35(3)-6

Kuzina V.V., Varentsov V.K., Koshev A.N., Kupriyanko G. Mathematical modeling and software for calculating regimes of galvanic wastewater purification from heavy and nonferrous metals in devices with flow-through three-dimensional electrodes. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 3, 2023. pp. 83-90.

Математическое моделирование и программное обеспечение для расчета режимов очистки гальванических стоков от тяжелых и цветных металлов в аппаратах с проточными трехмерными электродами

[1] *В.В. Кузина*, ORCID: 0000-0003-4511-7176 <*kuzina@pguas.ru*>
[2] *В.К. Варенцов*, ORCID: 0000-0001-8622-9364 <*vvk@ngs.ru*>
[1] *А.Н. Кошев*, ORCID: 0000-0003-3057-4980 <*koshev@pguas.ru*>
[1] *Г.М. Куприянко*, ORCID: 0009-0001-6737-554X <*gtaas@mail.ru*>

[1] *Пензенский государственный университет архитектуры и строительства, 440028, Россия, г. Пенза, ул. Германа Титова, 28.*
[2] *Институт химии твердого тела и механохимии СО РАН, 630128, Россия, г. Новосибирск, ул. Кутателадзе, 18*

Аннотация. В работе кратко обсуждаются прогрессивные технологии очистки сточных вод от ионов тяжелых и цветных металлов промышленных и малых предприятий городских агломераций. Приводится анализ эффективности применения проточных трехмерных электродов для очистки сточных вод от вредных реагентов. Приводятся математические модели электрохимических процессов в трехмерных проточных электродах применительно к извлечению металлов из растворов гальванохимических производств. Приводится описание комплекса программ, разработанных для проведения вычислительных экспериментов по полученным математическим моделям. При использовании программного комплекса получено численное решение научной задачи, имеющей практическое значение. Показано хорошее соответствие результатов расчетов и экспериментов.

Ключевые слова: математическое моделирование; комплекс программ; язык программирования Object Pascal; проточный трехмерный электрод; извлечение металлов из растворов гальванохимических производств.

Для цитирования: Кузина В.В., Варенцов В.К., Кошев А.Н., Куприянко Г.М. Математическое моделирование и программное обеспечение для расчета режимов очистки гальванических стоков от тяжелых и цветных металлов в аппаратах с проточными трехмерными электродами. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 83–90 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–6

1. Introduction

A new approach to the study of electrochemical processes in three-dimensional flowing electrodes as applied to the extraction of metals from solutions of galvanochemical production is to conduct computational experiments based on mathematical modeling and software package.

Galvanochemical productions, with all their advantages, have a sufficient environmental hazard, which is mainly determined by the harmful impact of galvanochemical effluents containing components of technological solutions on surface and ground water bodies, including in urban agglomerations. The exceeding of the maximum permissible concentrations (MPC) of metals in wastewater is caused by salvo discharges of electrolytes in electroplating plants. The problem of salvo discharges is caused by so-called "technical fatigue" of solutions. Such discharges lead to high water consumption, disruption of treatment facilities, a sharp increase in MPC in wastewater. The causes of deterioration of electrolytes are usually associated with the accumulation of inorganic and organic substances in them, including impurities introduced with reagents, products of electrode reactions. Conditioning of electrolytes, extraction of valuable components of electrolytes, and their reuse is one of the possible ways to solve the problems of creating closed technological processes.

Until recently, in the treatment of galvanic and other wastewater the dominant direction was the treatment of the total effluent of galvanochemical production [1]. However, there are a number of publications showing that the creation of local solution processing systems gets the greatest

application, as local cycles along with the solution of environmental problems provide the return of reagents and water, allowing the creation of low- and waste-free production [2, 3].

2. Methodology

Among the known methods of creating local treatment systems for galvanic solutions a special place should be given to electrochemical methods of extraction of valuable components to return them into industrial cycles. Electrochemical methods satisfy the basic requirements to the processes developed for extraction of metals from solutions of galvanochemical productions: they do not require the use of reagents, the metal is obtained in the purest concentrated form and can in most cases be returned to production; the possibility of process automation is easily realized, water consumption is reduced [4–7], etc.

One of the promising methods for solving this problem seems to be the use of apparatuses with flow-through three-dimensional electrodes (FTE) for extracting metals from solutions with low concentrations. The development of original technologies of electrodeposition of various metals based on the use of FTE is necessary for the intensification of electrochemical processes, especially in solutions with low concentrations of electroactive components, which is achieved primarily through the use of cathode materials with a high reaction surface and the possibility of intense mass transfer in the electrode volume [4–7]. The solution of this problem is obviously promoted by the development of methods of mathematical modeling of processes in FTE and application of computational experiment. The use of mathematical modeling makes it possible to calculate and predict the results of the technological process, as well as to optimize the process by calculating the effective values of control parameters [4–6, 8].

Let's assume that the flux of charged particles of the i-th grade N_i (i = 1, ..., n) in the electrolyte volume is determined by migration and convective components, which is realized in most electrochemical systems [9]:

$$N_i = z_i u_i F C_i \nabla E + C_i V. \quad (1)$$

Here z_i, C_i, u_i are, respectively, the charge, concentration and mobility of the i-th electroactive component in a pseudo-homogeneous medium; ∇E is the electric field potential gradient; V is the velocity vector of the convective transfer of the solution.

The current in the electrode-electrolyte volume is expressed by the formula:

$$j = F \Sigma(z_i N_i). \quad (2)$$

Material balance condition in the absence of a homogeneous electrochemical reaction:

$$\partial C_i / \partial t = -\nabla \bullet N_i. \quad (3)$$

Here $\nabla \bullet N_i$ is the divergence of the N_i flow.

Conversion of equations (1) – (3) using known rules of differential calculus, as well as the equation of the relationship between the change in concentration of the extractable component C_k with the partial current density j_{Sk} [10]

$$\partial C_k / \partial r = - S j_{Sk} / (|V| z_k F) \quad (4)$$

eventually leads to the following system of differential equations:

$$F \partial \Sigma(z_k C_k) / \partial t = \nabla \bullet [-\kappa_s \kappa_l \nabla E / (\kappa_s + \kappa_l)] + S \Sigma(j_{Sk}). \quad (5)$$

Here S is the reaction surface; j_{Sk} is the polarizing current density by k-component, κ_s, κ_l – conductivities of solid and liquid phases of the system. Together with natural boundary conditions:

$$[\partial E(t)/\partial n]_{\sigma c} = j(t)\rho_s, \ [\partial E(t)/\partial n]_{\sigma a} = j(t)\rho_l \quad (6)$$

$$[\partial E(t)/\partial n]_{\sigma i} = 0, \ [\partial E(t)/\partial n]_{\sigma e} = C_0. \quad (7)$$

Here n is the direction of the normal to the boundary of the reaction region, consisting of the surfaces of the cathode, anode, insulators and the electrolyte supply zone: $\sigma = \sigma_c + \sigma_a\ \sigma_i + \sigma_e$; ρ_s, ρ_l are the specific resistance of the solid and liquid medium, respectively.

The system (5) – (7) makes it possible to calculate the distribution of potential, current density, and concentration of the electroactive substance in the volume of the porous electrode.

The mathematical models given in [4-7] are based on the development of earlier modeling results obtained both by the authors of the present paper and by other authors, for example, in [11], but the generalized dynamic models proposed by the authors of the present paper are not found in the literature.

The study and criticism of the mathematical models of the authors of this publication and other authors are given in detail, for example, in [12].

Mathematical aspects of the implementation of the proposed mathematical models of electrodeposition in three-dimensional flow electrodes are discussed, for example, in [13].

To carry out computational experiments, a set of programs has been developed to simulate electrochemical processes in PTE [14]. The software package is designed to calculate and analyze the parameters of the electrochemical process when extracting metals from electrolyte solutions onto the flowing carbon-fiber electrode. The main structural elements of the program complex are modules: *general.pas* is the main form that allows opening the forms for entering basic process parameters – the *parametr.pas* module, for selecting electrochemical constants from available files – the *vybor.pas* module, for editing data – the *redact.pas* module, modules for calculating process parameters, optimization, plotting dependencies, etc. Program modules can be executed in parallel that allows increasing speed of computer calculations.

The program complex is implemented in the Object Pascal programming language [15-17]. Used libraries are the following: Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, Menus, StdCtrls, etc. An IBM PC-compatible PC with Windows XP/Vista/7/8/10 operating systems was used for the computational experiments.

Input data for calculation are parameters of the deposited component (valence of ions, diffusion coefficient), concentration of ions of the deposited component in solution, solution flow rate, mass transfer coefficient, electrical conductivity of solid and liquid phases of the electrode-solution system, overall current density, electrode parameters (porosity, fiber radius, electrode thickness), electrolysis time.

The calculation results are: metal ion concentrations (calculation can be performed for one or two metal ions contained in the solution), distribution over the thickness (volume) of the electrode of its conductivity, metal mass, electrolyte flow rate, metal ion mass transfer coefficients, mass transfer coefficient for oxygen, potential, metal current profiles, oxygen current profile, hydrogen current profile, reduced metal current profile.

One of the most recent works on the use of the software package developed by the authors is, for example, the article [18].

3. Results and discussion

The original approach consisting in the representation of the carbon-graphite cathode in the electrochemical reactor with the flowing three-dimensional electrode made of fibrous carbon materials in the form of pseudo-homogeneous medium has allowed to construct the dynamic mathematical models of the process of metals extraction from electroplating waste water and regeneration of spent electrolytes. Based on the models a set of programs in the Object Pascal programming language was developed that allows conducting theoretical research of regularities of the electrodeposition process and determining optimal parameters of the process and the electrolyzer to solve various technological problems. One of the program complex variants is published, for example, in [14].

To illustrate the efficiency of the methods let us calculate co-deposition of gold and silver from thiourea sulfate solution with the following composition: H_2SO_4 – 0.5 mol/l, thiourea – 50 g/l, gold – 22.4 mg/l, silver – 141 mg/l (*Fig. 1*).

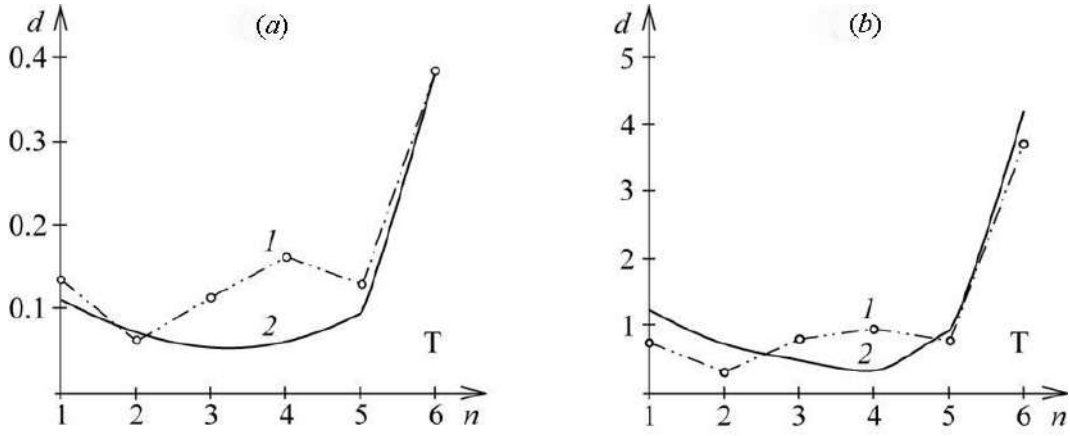


Fig. 1. Distribution of metal sludge over the thickness of the electrode: (a) – gold; (b) – silver; d – metal mass to CFM mass ratio; 1 – experiment; 2 – calculation; n – layer number; T – electrode back side; electrode thickness – 1 cm; current density – 0.2 A/m^2 , the solution flowrate – 0.56 cm/s.

The studies were carried out with frontal (from the side of the counterelectrode) feeding of the solution into the electrode. In the first case, the electrode was composed of 12 layers of carbon fiber material (CFM) and 6 layers in the second case. The characteristics of the CFM (grade VVP-66-95) are the following: specific surface area – 255 cm^2/cm^3, specific conductivity – 0.03 siemens/cm, porosity – 0.95 [4–6]. The specific electrical conductivity of the solution – 0.1 siemens/cm, electrolysis time – 60 minutes.

The experimental and calculated dependences presented in *Fig. 1*, as well as the consistency of the results of calculations of electrochemical functions of classical electrochemical theory allow us to conclude that the mathematical models and calculation algorithms described in this communication and in our other works [4–6] are effective for numerical studies and optimization of the control parameters of processes of metal ions extraction from industrial waste water to flowing threeedimensional electrodes for the purpose of decontaminating waste water. It should be noted that FTEs could be effectively used simultaneously for cathodic metal extraction and anodic oxidation of toxic electrolyte compounds in one electrolyser [4–7].

4. Conclusions

The application of mathematical modeling for the study of electrochemical processes in three-dimensional flowing electrodes in order to solve the problem of metal extraction from solutions of galvanochemical industries allows us to calculate and predict the results of the technological process, as well as to optimize the process by calculating the effective values of the control parameters. The use of the developed program complex allowed to obtain a numerical solution of the scientific problem of practical importance. A good correspondence of calculation and experimental results is shown. The possibility of parallel execution of software modules allows increasing the speed of computer calculations.

References

Свергузова С.В. Комплексное обезвреживание сточных вод, утилизация осадков водоочистки и вторичное использование гипсо- и металлсодержащих промышленных отходов. 2008. Автореферат докторской диссертации, КГТУ, г. Казань, 38 с. / Sverguzova S.V. Integrated wastewater neutralization, utilization of water treatment sludge and recycling of gypsum and metal-containing industrial waste, 2008. Author's abstract of doctoral dissertation, KSTU, Kazan, 38 p. (in Russian).

Арутюнян Д.М., Трегубов А.Ю. Способ очистки промывных вод и отработанных электролитов. 2017. В сборнике трудов конференции, Волгоградский государственный технический университет, Волгоград, стр. 81-83. Arutyunyan D.M., Tregubov A.Yu. Method for purification of wash water and spent electrolytes. In the Proceedings of the conference, Volgograd State Technical University, Volgograd, 2017, pp. 81-83. (in Russian).

Синельцев А.А. Сорбционная очистка сточных вод от ионов тяжелых металлов с помощью модифицированного гранулированного глауконита. 2016. Автореферат кандидатской диссертации, Казан. нац. исслед. технол. ун-т., Казань, 22 с. / Sineltsev A.A. Sorption purification of wastewater from heavy metal ions using modified granulated glauconite, 2016. Ph.D. thesis abstract, Kazan National Research Technological University, Kazan, 22 p. (in Russian).

Варенцов В.К., Кошев А.Н., Варенцова В.И. Современные проблемы электролиза и задачи оптимизации процессов в реакторах с трехмерными углеродными электродами. Пенза: ПГУАС, 2015. 284 с. / Varentsov V.K., Koshev A.N., Varentsova V.I. Modern problems of electrolysis and optimization problems of processes in reactors with three-dimensional carbon electrodes: monograph. Penza: PSUAC, 2015, 284 p. (in Russian).

Варенцов В.К., Кошев А.Н., Варенцова В.И., Кузина В.В. Окислительно-восстановительные процессы на проточных трехмерных электродах. Математическое моделирование. Теория. Эксперимент. Пенза: ПГУАС, 2020, 172 с. / Varentsov V.K., Koshev A.N., Varentsova V.I., Kuzina V.V. Oxidation-reduction processes on flowing three-dimensional electrodes. Mathematical modeling. Theory. Experiment: monograph. Penza: PSUAC, 2020, 172 p. (in Russian).

Кузина В.В., Варенцов В.К., Кошев А.Н. Теория, математическое моделирование и экспериментальные исследования электрохимических процессов в проточных трехмерных электродах при электроосаждении металлов из растворов гальванических производств. Пенза: ПГУАС, 2022, 204 с. / Kuzina V.V., Varentsov V.K., Koshev A.N. Theory, mathematical modeling, and experimental studies of electrochemical processes in flowing three-dimensional electrodes during electrodeposition of metals from galvanic solutions: monograph. Penza: PSUAC, 2022, 204 p. (in Russian)

Варенцов В.К. Электролиз с трехмерными электродами в процессах регенерации металлов из промывных растворов гальванических производств / Известия Сибирского отделения Академии наук Союза Советских социалистических республик (СО АН СССР). Сер. химических наук. 1988. Вып. 3. Стр. 124-138. / Varentsov V.K. Electrolysis with three-dimensional electrodes in the processes of metal regeneration from washing solutions of galvanic production / Proceedings of the Siberian Branch of the AS USSR. Ser. of Chemical Sciences, 1988, vol. 3, pp. 124-138. (in Russian).

Скороходов В.Ф., Китаева А.С., Бирюков В.В., Никитин Р.М., Артемьев А.В. Компьютерное моделирование процессов очистки промышленных сточных вод для выбора их оптимальных параметров. 2019. В сборнике трудов конференции "Теория и практика системной динамики". Кольский научный центр Российской академии наук (Апатиты), апрель 2019. стр. 137-139. / Skorokhodov V.F., Kitaeva A.S., Biryukov V.V., Nikitin R.M., Artemyev A.V. Computer modeling of industrial wastewater treatment processes to select their optimal parameters. In Proceedings of the conference "Theory and practice of system dynamics". Kola Scientific Center of the Russian AS (Apatity), April, 2019, pp. 137-139. (in Russian).

Ньюмен Дж. [Newman J.] Электрохимические системы. М.: Мир, 1977. 463 с. / Newman J. Electrochemical Systems. Moscow: Mir, 1977, 463 p. (in Russian).

Sioda R.E. Flow through electrodes composed of parallel screens. Electrochem Acta, 1977, vol. 22, nom. 4, pp. 439–443.

Маслий А.И., Поддубный Н.П., Медведев А.Ж. Влияние скорости и направления протока раствора на осаждение металла внутри пористого электрода. Конечная масса осадка и его распределение. Электрохимия, 2006. Т.42, №2. Стр. 183-189. / Masliy A.I., Poddubny N.P., Medvedev A.J. The influence of the speed and direction of the flow of the solution on the deposition of metal inside the porous electrode. The final mass of the sediment and its distribution / Electrochemistry, 2006, vol. 42, nom. 2, pp. 183-189. (in Russian).

Кошев А.Н., Варенцов В.К., Чиркина М.А. Анализ математических моделей и теория распределения поляризации проточных объемно-пористых электродов. Физикохимия поверхности и защита материалов. 2009. Т. 45. № 4. Стр. 441-448. / Koshev A.N., Varentsov V.K., Chirkina M.A. Analysis of mathematical models and polarization distribution theory of flow-through porous electrodes / Fizikokhimiya surface and protection of materials, 2009. vol. 45, nom. 4, pp. 441-448. (in Russian).

Кошев А.Н., Варенцов В.К., Чиркина М.А., Камбург В.Г. Математическое моделирование и теория распределения поляризации в электрохимических реакторах с проточными объемно-пористыми катодами/ А. Н. Кошев, // Математическое моделирование. 2011. Т. 23. № 8. Стр. 110-126. / Koshev A.N., Varentsov V.K., Chirkina M.A., Kamburg V.G. Mathematical modeling and theory of polarization distribution in electrochemical reactors with flow-through volume-porous cathodes / Mathematical modeling, 2011, vol. 23, nom. 8, pp. 110-126. (in Russian).

Гвоздева И.Г., Кошев А.Н., Воронцов А.А. Программный комплекс для расчета параметров электрохимического процесса в проточных углеродных волокнистых электродах. Свидетельство о регистрации программы для ЭВМ 2022616439, 08.04.2022. Заявка № 2022614434 от 24.03.2022. / Gvozdeva I.G., Koshev A.N., Vorontsov A.A. Software complex for calculating the parameters of the electrochemical process in flowing carbon fiber electrodes. Registration certificate of software 2022616439, 08.04.2022. Application number 2022614434 of 24.03.2022. (in Russian).

Official website of the Free Pascal compiler. [Electronic resource] Access mode: https://www.freepascal.org/ (Accessed 02/15/2023).

Official website of the GMP library. [Electronic resource] Access mode: https://www.gmplib.org/ (Accessed 20.02.2023).

Жорняк А.Г., Морозова Т.А. Применение среды разработки LAZARUS для научных и инженерных исследований. Часть I. Математические вычисления и визуализация полученных результатов. Научно-технический вестник Поволжья. 2023. № 1. С. 56-59. / Zhornyak A.G., Morozova T.A. Application of the LAZARUS development environment for scientific and engineering research. Part I. Mathematical calculations and visualization of the results obtained. Scientific and technical bulletin of the Volga region. 2023. No. 1. pp. 56-59.

Варенцов В.К., Кошев А.Н. Теоретические и экспериментальные исследования эффекта образования анодных зон в объеме катодно-поляризованного трехмерного электрода из углеродного волокнистого материала. // Теоретические основы химической технологии. 2022. Т. 56. № 4. Стр. 464-473. / Varentsov V.K., Koshev A.N. Theoretical and experimental studies of the effect of anodic zones formation in the cathodically polarized three-dimensional electrode made of carbon fiber material. / Theoretical basis of chemical technology, 2022, vol. 56, nom. 4, pp. 464-473. (in Russian) DOI: 10.31857/S0040357122040078.

Информация об авторах / Information about authors

Валентина Владимировна КУЗИНА – кандидат технических наук, доцент, доцент кафедры «Информационно-вычислительные системы» Пензенского государственного университета архитектуры и строительства. Сфера научных интересов: информационные технологии и системы, программирование, математическое моделирование, численные методы.

Valentina Vladimirovna KUZINA – Candidate of Technical Sciences, Associate Professor, Associate Professor of the Department "Information and Computer Systems" at Penza State University of Architecture and Construction. Research interests: information technologies and systems, programming, mathematical modeling, numerical methods.

Валерий Константинович ВАРЕНЦОВ – доктор технических наук, профессор, ведущий научный сотрудник лаборатории гетерогенных систем Института химии твердого тела и механохимии СО РАН. Сфера научных интересов: теоретические основы технологических процессов и электрохимических реакторов с проточными трехмерными электродами, разработка технологий и электролизеров для электроосаждения металлов из растворов, интенсификации окислительно-восстановительных процессов.

Valery Konstantinovich VARENTSOV – Doctor of Technical Sciences, Professor, leading research associate of the Laboratory of heterogeneous systems at the Institute of Solid State Chemistry and Mechanochemistry SB RAS. Research interests: theoretical bases of technological processes and electrochemical reactors with flow-through three-dimensional electrodes, development of technologies and electrolyzers for electrodeposition of metals from solutions and intensification of redox processes.

Александр Николаевич КОШЕВ – доктор химических наук, профессор, профессор кафедры «Информационно-вычислительные системы» Пензенского государственного университета архитектуры и строительства. Сфера научных интересов: информационные технологии, математическое моделирование и оптимальное управление технологическими процессами в строительстве, электрохимии, экологии.

Alexander Nikolaevich KOSHEV – Doctor of Chemical Sciences, Professor, Professor of the Department of information and computer systems at Penza State University of Architecture and Construction. Research interests: information technologies, mathematical modeling and optimal control of technological processes in construction, electrochemistry, ecology.

Глеб Михайлович КУПРИЯНКО – аспирант кафедры «Информационно-вычислительные системы» Пензенского государственного университета архитектуры и строительства. Сфера научных интересов: информационные технологии и системы, программирование.

Gleb Mikhailovich KUPRIYANKO – postgraduate student of the Department "Information and Computer Systems" Penza State University of Architecture and Construction. Sphere of academic interests: information technologies and systems, programming.

DOI: 10.15514/ISPRAS-2023-35(3)-7

"Symcrete" Memory Model with Lazy Initialization and Objects of Symbolic Sizes in KLEE

[1] *S.A. Morozov,* ORCID: 0000-0003-1160-5614 <*morozov.serg901@gmail.com*>
[2] *A.V. Misonizhnik,* ORCID: 0000-0002-5907-0324 <*misonijnik@gmail.com*>
[3] *D.A. Mordvinov,* ORCID: 0000-0002-6437-3020 <*mordvinov.dmitry@gmail.com*>
[3] *D.V. Koznov,* ORCID: 0000-0003-2632-3193 <*d.koznov@spbu.ru*>
[4] *D.A. Ivanov,* ORCID: 0000-0002-0420-9077 <*korifey@gmail.com*>

[1] *National Research University 'Higher School of Economics',*
16, Soyuza Pechatnikov Street, Saint Petersburg, 190121, Russia.
[2] *IT Solutions Inc.,*
41, Novoslobodskaya Street, Moscow, 127055, Russia.
[3] *St. Petersburg State University,*
7/9, Universitetskaya Embankment, Saint Petersburg, 199034, Russia.
[4] *Huawei Technologies Co., Ltd.,*
69-71, Marata Street, Saint Petersburg, 191119, Russia

Abstract. Dynamic symbolic execution is a well-known technique for testing applications. It introduces symbolic variables – program data with no concrete value at the moment of instantiation – and uses them to systematically explore the execution paths in a program under analysis. However, not every value can be easily modelled as symbolic: for instance, some values may take values from restricted domains or have complex invariants, hard enough to model using existing logic theories, despite it is not a problem for concrete computations. In this paper, we propose an implementation of infrastructure for dealing with such "hard-to-be-modelled" values. We take the approach known as symcrete execution and implement its robust and scalable version in the well-known KLEE symbolic execution engine. We use this infrastructure to support the symbolic execution of LLVM programs with complex input data structures and input buffers with indeterminate sizes.

Keywords: symbolic execution; software analysis; lazy initialization; symcrete execution; smt-solvers.

For citation: Morozov S.A., Misonizhnik A.V., Mordvinov D.A., Koznov D.V., Ivanov D.A. "Symcrete" memory model with lazy initialization and objects of symbolic sizes in KLEE. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 91-108. DOI: 10.15514/ISPRAS-2023-35(3)-7

Acknowledgements. This work is supported by the grant of the Russian Science Foundation (RSF) № 22-21-00697.

Симкретная модель памяти с ленивой инициализацией и объектами символьного размера в символьной виртуальной машине KLEE

[1] *С.А. Морозов,* ORCID: 0000-0003-1160-5614 <*morozov.serg901@gmail.com*>
[2] *А.В. Мисонижник,* ORCID: 0000-0002-5907-0324 <*misonijnik@gmail.com*>
[3] *Д.А. Мордвинов,* ORCID: 0000-0002-6437-3020 <*mordvinov.dmitry@gmail.com*>
[3] *Д.В. Кознов,* ORCID: 0000-0003-2632-3193 <*d.koznov@spbu.ru*>
[4] *Д.А. Иванов,* ORCID: 0000-0002-0420-9077 <*korifey@gmail.com*>

[1] *Национальный исследовательский университет “Высшая школа экономики”,*
Россия, 190121, Санкт-Петербург, Союза Печатников ул., д.16.
[2] *IT Solutions Inc.,*
Россия, 127055, Москва, Новослободская ул., д.41.
[3] *Санкт-Петербургский государственный университет,*
Россия, 199034, Санкт-Петербург, Университетская наб., д. 7-9.
[4] *Huawei Technologies Co., Ltd.,*
Россия, 191119, Санкт-Петербург, Марата ул., д. 69-71.

Аннотация. Динамическое символьное выполнение – хорошо известный метод тестирования приложений. Он вводит понятие символьной переменной – данных программы, не имеющих конкретного значения в момент объявления, – и использует их для систематического изучения путей выполнения в анализируемой программе. Однако не **Санкт-Петербургский государственный университет** каждое значение может быть легко смоделировано как символическое: например, некоторые значения могут принимать ограниченное число значений или иметь сложные инварианты, которые достаточно сложно смоделировать с использованием существующих логических теорий несмотря на то, что это не является проблемой для конкретных вычислений. В этой статье мы предлагаем реализацию инфраструктуры для работы с такими “трудно моделируемыми” значениями. Мы используем подход, известный как симкретное исполнение, и реализуем его надежную и масштабируемую версию в хорошо известном движке символьного выполнения KLEE. Мы используем эту инфраструктуру для поддержки символьного исполнения программ на языке LLVM со сложными структурами входных данных и входными буферами неопределенных размеров.

Ключевые слова: символьное исполнение; анализ программного обеспечения; ленивая инициализация; симкретное исполнение; smt-решатели.

Для цитирования: Морозов И.А., Мисонижник А.В., Мордвинов Д.А., Кознов Д.В., Иванов Д.А. Симкретная модель памяти с ленивой инициализацией и объектами символьного размера в символьной виртуальной машине KLEE. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 91–108 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–7

Благодарности. Работа поддержана грантом Российского научного фонда (РНФ) № 22-21-00697.

1. Introduction

Dynamic symbolic execution is a software testing technique that allows exploring execution paths in a program under analysis, generates test coverage, and finding bugs in a given source code (e.g. out of bound memory errors or signed integer overflows) [1]. This is done by marking some program variables as symbolics, in other words, variables with no specific value. During analysis, a symbolic engine adds logical constraints to them, which possibly restrict values in different paths. To prove the satisfiability or unsatisfiability of a set of constraints, symbolic engines widely use SMT-solvers [2], such as Z3 [3], CVC5 [4], bitwuzla [5] and many others.

Encoding a set of values with logical constraints for each symbolic variable is one of the crucial ideas in symbolic execution. This approach enables keeping several program executions as a single execution state at the current position in the exploration path. All possible solutions for these constraints then become the values of symbolic variables in corresponding execution states. Since solving such formulas is an NP-hard problem, the performance and completeness of the solution heavily rely on the number and size of the logical formulas passed to the SMT-solver.

However, some values in a program can be hard to model by decidable logical constraints. The problem arises from the fact that the values of a variable may belong to a restricted domain. Such domains can have implicit and complex rules to encode in a logical formula. Let us provide some examples in which the described problem appears:

- *Objects with symbolic sizes*. Program under analysis may dynamically allocate memory on the heap (e.g. with `malloc(n)` in C language or `operator new[n]` in C++). If we treat the argument passed to that function as symbolic, we will allocate an object whose size may have different values depending on the current execution path (object with *symbolic size*). Consider an example presented in *Listing 1*.

```
int foo (int n) {
    char * s = (char *) malloc (n);
    if (n == 1) {
       s [0] = 0;
    } else if (n > 1) {
       s [1] = 10;
    }
    return 1;
}
```

Listing 1. Dynamic allocation

  If we pass a symbolic argument to that function, we will allocate an object with symbolic size at the first line. Then the allocated object will have different sizes at the distinct branches of `if`-statement. Modelling objects with symbolic size might take many computational resources. Each allocated memory object is represented as a separate entity and cannot intersect with other objects. Naïve modelling of these restrictions may result in SMT solvers needing to handle $O(n^2)$ constraints, where n is the number of memory objects. Such modelling can significantly impact the performance of symbolic execution.

- *External calls*. During program exploration, the symbolic execution engine may meet calls to undefined or external functions, i.e. functions with no sources provided. As the engine does not have any information about the encountered function, it cannot properly model function behaviour to continue accurate analysis: for instance, the return value of this function may take a limited number of values. Interpreting return value as a symbolic value may be too excessive to model function behaviour, and the symbolic engine is doomed to lose precision in this case.

One possible behavior is plain modelling of all such behaviors described in the bullets above. In this case, the engine *over-approximates* program behavior, i.e. explores more paths than there are. Therefore, it degrades performance and accuracy.

Another behavior, taken, for instance, in KLEE symbolic execution engine [1], is to fix one possible solution during analysis. When the engine meets specific code constructions, it picks up the solution for all symbolic variables involved in one. Then it restricts taken variables with values

from the received concrete solution for the following exploration. For instance, the constructs described above are modelled as follows:

- *Objects with symbolic sizes*. We might avoid performance issues by choosing one exemplar of a symbolic size fitting current constraints at the moment of the allocation. For example, while executing the `malloc(n)` statement at *Listing 1*, KLEE would choose some concrete value of `n` fitting the current path constraints, say, `n = 1`. But then, branchings on `n` would be evaluated only within this concrete assignment, leading to missed branches. In this case, KLEE misses covering the `s[1] = 10;` statement.
- *External calls*. Calls to external or undefined functions may be modelled as actual calls to these functions. As such functions might take arguments, which were marked as symbolic variables before, the symbolic execution engine needs to find a solution for them to satisfy previously added logical constraints. Return value then will be a constant value and cannot be treated as a symbolic value.

In these cases, the engine explores fewer paths than actually exist. On the one hand, it leads to performance improvements, as the engine analyses a smaller number of possible program behaviours. On the other hand, it impairs the engine's ability to find vulnerabilities in a program under analysis, leading to a non-exhaustive search through the program inputs space. In other words, this approach *under-approximates* program behaviors.

The idea that can be applied to resolve problems discussed above is to use a well-known approach of *symcrete*[1] [6, 7] execution. This feature allows a symbolic execution engine to mark variables as symbolic, but additionally keep a concrete value (*concretization*) for it satisfying some set of logical constraints. This concretization might be given by an algorithm different from the SMT-solver. Therefore, if such algorithms maintain some invariants inside, then they will be automatically satisfied for produced models.

The described idea gives several opportunities to the KLEE execution engine, but one of the most interesting is the support of objects with indeterminate sizes. It is achieved due to the property of allocators to allocate non-intersecting objects and the property of symcretes to keep concrete values fitting current constraints. Hence, we can dynamically maintain memory layout with no significant impact on performance. The feature of objects with symbolic sizes would increase the engine's precision for detecting buffer overflows and other memory issues in LLVM programs.

Symcretes should be fully compatible with the existing features of the symbolic virtual machine, such as lazy initialization [8, 9]. This technique enables the exploration of program behaviors with complex input data structures.

In summary, the main contributions of this paper are:

1) Implementation of the infrastructure of symcrete execution in KLEE.
2) Application of this infrastructure to model objects of symbolic sizes.
3) Application of this infrastructure to improve the currently existing mechanism of lazy initialization.

2. Background

Before discussing the main ideas of this paper, let us introduce the basic concepts of symbolic execution used throughout this paper.

[1] "Symcrete" = symbolic + concrete.

2.1 Execution and forking

Dynamic symbolic execution executes a program with *symbolic* variables, i.e. values that represent all possible concrete program inputs. During program exploration, the execution engine operates with *execution states*, which can step over one instruction and fork. For these states, the symbolic execution engine maintains the inner representation of programs *memory model*. Also, every execution state maintains path constraints (PC) – a set of logical formulas describing the explored path. When the execution engine meets a conditional operator, it queries the solver with constraint and its negation, and forks state if solutions for both constraints exist. If only one statement is true, it does not fork and simply proceeds the execution of a reachable path.

Take a look at the example in *Listing 1*: let `n` be a symbolic parameter of the function. In the beginning, path constraints are empty, and the inner memory representation contains only one record: $n \leftarrow \lambda$. After execution state meets the line `if (n == 1) { ... }`, it queries solver about the *validity* of PC with $\lambda = 1$ and PC with $\neg(\lambda = 1)$. As they are both satisfiable, it splits the current execution state into two states with the same objects in memory and path constraints $PC' = PC \wedge \lambda = 1$, $PC'' = PC \wedge \neg(\lambda = 1)$ correspondingly.

2.2 Memory model

Objects in memory have addresses, which represent their location in the symbolic engine's address space, sizes, representing the number of allocated bytes for their content in address space, alignment, which makes restrictions on an address (for instance in source code user can call `posix_memalign` and `memalign` functions), and contents, an array of (potentially symbolic) bytes. To handle all that information, symbolic engines maintain *memory model*, which stores required information about all currently existing objects: addresses, sizes, contents, and so on.

2.3 Constraints Representation

Every constraint in KLEE is an *expression*. Expression is a tree, each node of those is an operation, and children are operands. Every leaf of these trees is either constant or read from a *symbolic array*. A symbolic array is an array from the SMT theory of arrays, i.e. unbounded storage of symbolic integers, supporting both load and store operations. Each store operation creates a new version of an array with a value changed by a specified index, therefore arrays can be considered immutable.

For brevity, we use the term "array" instead of "symbolic array".

2.4 Validity Cores

A set of constraints with a statement may be *valid*, that is, no counterexample can be found for it, and *invalid* otherwise. To check the validity of expressions, the engine queries SMT-solver with a given set of assumptions and negation of the provided statement. If SMT-solver gives a solution that satisfies the received query, then a counterexample is found and the initial statement in the assumption of constraints from the set is invalid. Otherwise, it may return a *validity core*, a subset of constraints "explaining" the validity.

For instance, consider the set of assumptions $\{\lambda < 10, \lambda > \alpha\}$ and a statement $\lambda > 10$. We would like to check the validity of a statement within the assumptions, that is, the validity of the formula

$$\forall \lambda, \alpha : \lambda < 10 \wedge \lambda > \alpha \Longrightarrow \lambda > 10$$

To show it, we might prove that the negation is unsatisfiable, i.e.

$$\exists\, \lambda, \alpha : \lambda < 10 \wedge \lambda > \alpha \wedge \neg(\lambda > 10)$$

SMT-solver would find a satisfying assignment, for example, $\{\lambda\, 1 \rightarrow 1, \alpha\, 1 \rightarrow 0\}$. It means that we have found a counterexample for the initial statement.

In contrast, if we check a statement $\lambda < 11$ with the same assumptions, we would query the satisfiability of $\{\lambda < 10, \lambda > \alpha, \neg(\lambda < 11)\}$ and receive from SMT solver the “unsatisfiable” verdict. State-of-the-art SMT solvers can compute unsatisfiable cores, a subset of conflicting statements. In this case, one unsatisfiable core is $\{\lambda < 10, \neg(\lambda < 11)\}$. It can be converted to validity core: just take assumptions from the unsatisfiable core as-is, and convert the negated statements from the unsatisfiable core to the original ones. In our example, the validity core includes the assumption $\lambda < 10$ and the statement $\lambda < 11$.

2.5 Optimizing solvers

As mentioned above, solving logical formulas, which have been constructed during program analysis, is the NP-hard problem. Hence, the complexity of the formulas in the query and the number of such queries becomes a bottleneck of symbolic execution. To simplify the queries to the solver, execution engines apply many optimizations for logical constraints. One way to provide such optimizations is to use optimizing solvers – solvers that can modify, separate, construct additional logical formulas, or even resolve received queries without calling an expensive SMT-solver. Such a solvers can form a chain ending with the SMT-solver.

2.6 Pointer resolution

Many languages, like C or C++, allow storing addresses directly into locations and dereference them. The resolution of concrete pointers is trivial, but symbolic execution engines might encounter programs with symbolic pointers. Consider the example in *Listing 2*.

```
int x = 10;
int y = 20;
void bar (int * s) {
   * s = 0;
}
```

Listing 2. Pointer resolution

As we do not know, at what address pointer `s` should be resolved, we must check every possible memory object, including the pointer variable itself. To handle these cases, the vanilla KLEE engine makes a pointer resolution operation: it iterates over all existing memory objects in memory and attempts to dereference given pointer into them: query the solver if a formula $ptr + idx > address \cap ptr + idx + type\_size < address + size$, with the formulas from path constraints, where ptr is a dereferencing pointer, idx is a relative offset (e.g. if we access the array by some index, `ptr[10]` in C or C++ languages), $type\_size$ is the size of the type we are trying access through, $address$ is the address of the memory object we are trying to access, $size$ is the size of that memory object. If the pointer can be dereferenced to the chosen memory object, KLEE forks the current execution state and modifies path constraints PC' of the received state with the above constraint.

In the example in *Listing 2*, pointer `s` can be resolved to at least two existing objects: `x` or `y`. After storing operation `*s = 0;` KLEE will maintain at least two execution states, in which `0` is written to `x` or `y`.

2.7 Lazy initialization

However, pointer resolution might not be enough to model all possible execution paths in a program. Suppose, you need to test a code for a linked list presented in *Listing 3*.

```
typedef struct Node {
    int x;
    Node * next;
}  Node;
int baz (Node l) {
    l.next -> x = 1;
    assert ((l.x + l.next -> x) % 2 ==0);
}
```

Listing 3. Linked list

In this code snippet `struct Node` contains a pointer to the next element in the linked list, which will be a symbolic value if we pass a symbolic argument to function `baz`. Consequently, pointer resolution at the line `l.next->x = 1` will proceed for the symbolic pointer in the same manner as described above. As we do not have any other objects of type `struct Node`, this code example will only test circular linked lists at most of length 1.

The problem here arises from the fact, that analyzing program does not contain explicitly initialized additional linked list nodes. We will face a similar problem if we try to analyze any recursive data structures, like Binary Search Trees, Linked Lists, and so on.

To overcome described obstacle modern symbolic engines apply a technique called *lazy initialization*. This method allows initializing additional objects in memory, if so required, to explore more program behaviors. Return to the example at *Listing 3*: during pointer resolution the symbolic execution engine will allocate one more additional object of type `struct Node` to model linked list with length at least 2 and fail the assertion `assert((l.x + l.next->x) % 2 == 0);` (as for circular linked list we summed two equal numbers before).

3. Design principles

During infrastructure design, we agreed on a set of principles to create a maintainable and easily extensible framework. These principles are as follows: (a) clear separation of public and private interfaces, (b) recompute only the demanded values, and (c) concretization should always exist. Let's consider them in more detail.

a) Clear separation of public and private interfaces: One of the most important requirements for symcretes architecture was to keep the symcretes public interface as simple as possible. Thus, to prevent the developers from implementing complex logic in various spots of symbolic engine code, the public interface of symcretes infrastructure should only provide methods to add a symcrete value to the execution state and to receive a current concretization for symcrete. All the internal architecture of symcretes and any processing details made by its infrastructure should not be accessible from the symbolic engine code.

b) Recompute only demanded values: Since the symcrete variable is the symbolic variable paired with the concrete value fitting some constraint set, then this concrete value may become obsolete with the addition of a new statement. As it might be difficult to receive a new

model for all symcrete variables in such situations, we require recomputing concrete values only for symcretes, which affects the validity of the query.

c) Concretization should always exist: At every moment, we should be able to receive an actual model for the symcretes used in the current constraint set. In other words, symcretes architecture should be similar to the “Observer” pattern, where the observable object is the solver and it should provide a possibility to subscribe to the solver updates.

4. Implementation

We have built our implementation on top of the KLEE of version 2.3 [10].

Followed by the principles described above we have separated symcretes and internal mechanisms to handle them, which we called *concretizing solver*. In our implementation symcrete is a pair of an array and a concrete value. To make a symcrete expressions we assign a read from created array to that expression. Concretization of symcretes is represented by the map from such arrays to bits storages.

To distinguish different symcretes we equipped all arrays with a new characteristic – arrays sources. These sources should reflect how the current array was received. For instance, an array that has been made to handle the addresses of memory objects should differ from arrays that are used to handle the content of memory objects. Also, these sources can carry useful properties for algorithms, which are used to generate values for them. We will show the application of these properties below.

The main logic for symcretes located in concretizing solver. It is one of the optimizing solvers, that can modify and handle received queries properly. In particular, concretizing solver modifies each query with constraints over symcretes: it adds equalities in form of `(Eq (Read width 0 symcrete_array), Constant)`, where `Read width offset source` is the read expression of width `width` at offset `offset` and array `source` – and passes them to the underlying solver. However, such modifications are not enough to handle symcretes.

Let us consider the following example. Suppose, we have a symcretes values x and y with concretization $x = 5$, $y = 10$, query with the set of assumptions $[x \leq 10, y \leq 20]$ and the statement $x \leq y$. Concretizing solver at the preprocessing stage will make additional constraints $x = 5, y = 10$, and consequently, the query will transform into a new query with the set of assumptions $[x \leq 10, y \leq 20, x = 5, y = 10]$ and statement $x \leq y$. Note, that this query is valid according to “validity logic”, as to compute validity we negate the statement, which results in $x > y$. Existing concretization cannot satisfy all assumptions with negated statement.

Therefore, existing concretization might add constraints, which force a given theorem to become valid, despite the original query being invalid. To solve such a problem, we process a symcretes *relaxation* after receiving a valid response from the solver. Symcretes relaxation is the algorithm that aims to recompute values for symcretes to receive an invalid response if so exists.

To implement it according to our principles, we need to find all symcretes that have inappropriate values (see principle “Recompute only required values”). Such values may be found in the validity core, which might be received from the solver. For that purpose, we extended the interface of KLEE’s solver with functions that may return validity cores on valid responses. Since then, we can process a relaxation after receiving a valid response with current concretization.

The relaxation algorithm is provided in Algorithm 1. More detailed, the core part of the algorithm is located in the `do { ... } while(...);` loop. It firstly constructs a concretized query by adding equality constraints on symcretes (line 5) and queries the solver with this query (line 6). If the response is already invalid, the loop can be completed (lines 7-9), and all we need is to assign appropriate values to symcretes, which have lost concretizations (lines 24-30). Otherwise, we will look at the validity core from the valid response and collect all symcrete arrays, those

concretizations affected validity (this is done by collecting all arrays and filtering them by predicate `isSymcrete` at line 11). After that, we check if we removed concretization, which was not removed before (lines 15-17). If so, we continue the process. Otherwise, the current validity core proves, that the initial query is valid.

In the general case, the presented process can take more than one iteration. This might happen, as SMT-solver does not guarantee to return all unsatisfiable sets of formulas from the given query: usually, they return any set of formulas that cannot be satisfied.

Let's see that in the example. For instance, we have symcretes x and y with concretizations 0 and 1 correspondingly, and statement $[x < y]$. The concretized query will have a form of $[x < y, x = 0, y = 1]$. Then we will query the solver with the statement $x \leq 0$. According to "validity logic" query will transform to a set of formulas $[x < y, x = 0, y = 1, x > 0]$, which cannot be satisfied, and we can highlight at least three unsatisfiable subsets: $[x = 0, x > 0]$, $[x < y, x > 0, y = 1]$ and $[x < y, x = 0, y = 1, x > 0]$. SMT-solver can return any of these. If it returns the first subset, the algorithm will remove concretization only for x, but the query will remain valid. Then on the second iteration, the SMT-solver return the second subset of formulas from the presented subsets. Consequently, the algorithm will remove concretization for y and after that find a counterexample to the initial statement, say, $x = 1, y = 2$.

After removing all outdated concretizations for symcretes we need to assign new values to them. To do that we query the registered algorithms (lines 24-26). After receiving new concretizations, we check if the solution for the entire query invalidates the received statement in the assumption of the given constraint set. If still not, we admit that the query is valid (lines 28-30). This can happen when concrete values for symcrete variables received from registered algorithms cannot provide values invalidating the query.

If the statement in the assumption of a set of given constraints is provably invalid, i.e. has a counterexample, then we store concretizations of symcretes involved in that query in a *concretization manager*. The concretization manager is the structure that stores concrete values for symcretes for all encountered invalid queries. It may be accessed from the symbolic execution engine to get the current concrete value for symcrete.

If we want to add a constraint without an explicit call to a solver, then we may lose the record to the concretization manager. In this case, we need to update it manually from the code location where the constraint is added.

Summing up all implementation details and principles, in KLEE to mark a variable as symcrete we need to create a new array. For that array, we need to specify its source. For arrays with such a source, we need to provide an algorithm which will be used to generate concrete values. To access the concrete value of the symcrete variable we may query the concretization manager with the constraint set and statement we are interested in.

In the next sections, we will show how we can use symcretes to support objects of symbolic sizes and improve the existing mechanism of lazy initialization.

4.1 Properties of objects of symbolic size

Before discussing the implementation of objects with symbolic sizes we need to discuss some of their properties. As we said before, every object has 3 main parameters: address from enclosing address space, size, and content. The content of memory objects can be considered independently from address and size, therefore we will not take it into account in the reasoning below.

Algorithm 1 Relaxation algorithm

```
1: function RELAX(query, symcretes)
2:     relaxationProceeded ← true;
3:     removedSymcretes ← [];
4:     do
```

```
5:          concretizedQuery ← query, symcretes;
6:          resp ← SOLVER.CHECK(concretizedQuery);
7:          if RESP.ISINVALID() then
8:             break;
9:          end if
10:         relaxationProceeded ← false;
11:         validSymcretes ← RESP.VALIDITYCORE().ALLARRAYS().FILTER(isSymcrete);
12:         if (validSymcretes\symcretes).ISEMPTY() then
13:            break;
14:         end if
15:         relaxationProceeded ← VALIDSYMCRETES.INTERSECT(symcretes).ISEMPTY();
16:         removedSymcretes ← REMOVEDSYMCRETES.UNION(validSymcretes);
17:         symcretes ← symcretes \ validSymcretes;
18:   while relaxationProceeded;
19:
20:   if    ¬relaxationProceeded then
21:         return Valid
22:   end if
23:
24:   for   sym ∈ removedSymcretes do
25:         sym ← GETVALUEBYSOURCE(sym.source);
26:   end for
27:
28:   concretizedQuery ← query, symcretes
29:   resp ← SOLVER.CHECK(concretizedQuery)
30:   return RESP.VALIDITY()
31: end function
```

Firstly, we may suppose, that addresses of objects with symbolic size may be considered as symbolic values. The idea comes from the fact, that two allocations with different sizes at the same location in source code will likely receive different addresses.

Secondly, we may assume that the size and address of one object are dependent values, i.e. changing of object’s size may affect the address in the enclosing address space.

Also, we need present several requirements for our implementation:

1) it should allow to dynamically resize objects
2) if several states maintain the same objects with different actual sizes, they must appear identically
3) it should consume as less memory, as possible

The logic behind the first requirement can be seen in the example at *Listing 4*.

```
char * s = malloc(n);
if (n > 1) {
   if (n > 2) {
      s [n - 1] = 2;
   }
}
```

Listing 4. Reallocation

In the assumption of `n` to be a symbolic variable, at the first line, we allocate an object with symbolic size. The most inner `if`-statement must be reachable with the object of size at least 3 addressable by pointer `s`.

The second requirement says, that states containing the same object with different concretized sizes must keep its properties: ID, alignment, allocation site, address and size expressions, and so on.

This requirement arises from the fact, that all actions are done with the specified object, and its properties cannot be violated or become outdated. Hence, after state forks, we must be able to use old constraints with new ones to find a solution for addresses and sizes in different branches of execution.

The last requirement states, that our implementation should use as less memory as possible. More detailed, since SMT-solvers work with variables as with numbers without any additional information, they might give huge models for objects with symbolic size. That may cause performance issues. Another problem is that the test case, that the symbolic engine will generate to report a bug, also can be huge enough. Usually, users want to receive the smallest test case to find the issue, therefore, we need to take care of that requirement.

4.2 Implementation of objects of symbolic size

As noted above, addresses of objects with symbolic sizes may be considered symbolic. Also, in the Section 1, we have already noticed, that we can use symcrete variables in this case.

To use them we added a new array source, which we called `AddressSource` and an algorithm, that will be able to generate solutions for such arrays. We introduced an `AddressGenerator` interface for that purpose. It has only one method `allocate(addressArray, size)`. All the classes implementing `AddressGenerator` should provide appropriate (e.g. non-overlapping) addresses for specified address array `addressArray` from the arguments list each time the `allocate(addressArray, size)` method is called.

We implement this interface in `AddressManager` class, which provides an additional method `allocateMemoryObject(addressArray, size)`.

This class is used in both concretizing solver and the execution engine. On call to `allocate` it allocates the memory, and ceiling size to the nearest power of 2. Then it creates a new memory object, that should copy all properties of the already existing memory object, that utilizes the same array as the address array and caches created object. It is also optimized for multiple allocations. Therefore, if the solver requests a size less than at least one of the cached memory objects, then it will return it (that optimizes memory consumption). Note, that in the worst case, this manager will use $2M$ bytes of memory, there $M = 2^{\lceil \log_2 S \rceil}$ and S is the size of the biggest memory object. An approach with the powers of 2 for allocated sizes has been chosen not to change concretizations of addresses for all other states, that use the same memory object. This is because certain states may force expressions to take concrete values (for instance, during the execution of an external call), and changing of address value for a group of states will invalidate such states.

`allocateMemoryObject(addressArray, size)` method is used to receive a memory objects created at `allocate` method. These memory objects are required to update an address space of execution state after recomputation of concretization for symcretes in its path constraints.

Since now, as we can maintain objects with symbolic addresses, we may apply symcretes to handle the model for objects with symbolic size. For that, we introduce symcretes with array source *SizeSource*. Symcretes with such source will contain values, corresponding to the size of memory objects, and therefore, their sum should be minimized (as we said in the requirements above). We extended KLEE's solver interface with a minimization algorithm, that solves an optimization problem and computes minimal possible values for a expression. This is done by the binary search on the answer for a given expression with a set of given assumptions.

One more important thing about this implementation is that address symcrete cannot become the reason for symcretes recomputation. It means that if in the algorithm at the *Listing 5* we received an address symcrete as a symcrete with a non-appropriate value and did not receive the size symcrete for the same object, we will not recompute the address and size. This is done for reasons that as we are using the system's allocator, we are not able to choose the values for addresses and ourselves.

Hence, if some concretization for some addresses violates constraints, then it is likely constraints on addresses were added and we cannot continue analysis for that execution path (except null check, in our implementation it is checked separately). For now, we cannot handle such situations properly, but for real-world problems, it covers most of the use cases.

Let’s see an example presented in *Listing 5*. In this example, we dynamically allocate memory objects of size `n`. At the moment of allocation `n` might take any possible value of type `unsigned`, and we do not know the exact size of allocated objects. As we are applying a minimization strategy for objects of symbolic sizes, the minimal possible value for the size of allocated objects is 0. Hence, before first `if`-statement exact size of allocated memory object in address space of enclosing execution state will be 0, and we will have two known symcretes: size and address with concretizations `0` and `$(malloc(0))` (return value of call to `malloc` function), correspondingly, and $PC = [n = ssize]$. Condition in the first `if`-statement adds constraint on the symcrete address of allocated memory object. Since then, in the unsatisfiable core we will have two constraints: $[saddress = \$(malloc(0)), saddress < 10]$. As it contains only symcrete for address, we say that we are not able to do anything if the current model is inappropriate. To execute the next `if`-statement we need to discuss one more optimization.

It may turn out, that from the given constraints we can deduce, that the size of the objects is a huge enough number. At *Listing 5* size of the allocated object in the `then` branch of second `if`-statement might take values not less than 100001. If we try to get a model for such arrays in the execution engine, we will receive problems with performance and memory consumption. To solve such problems, we extended KLEE with structure *SparseStorage* — it is a byte buffer with the specified default value. To fill it we query the solver only about bytes in the array that were used for reads that were applied to receive a model within this query. Is allowed to greatly reduce memory usage and increase performance.

```
unsigned n <- symbolic;
char * s = (char *) malloc(n);
if (s < 10) {
   exit (1);
}
if (n > 100000) {
   printf ("Huge!");
} else {
   printf ("Small!");
}
```

Listing 5. Symbolic size allocation

Returning to the example, both branches of second `if`-statement are reachable with our execution state. In the `then` branch we will have an object of size 100001, and in the `else` branch — an object of size 0.

The last implementation detail is related to default values of uninitialized memory objects not marked as symbolic. In the real world almost always content of memory allocation consists of undefined bytes. In the initial KLEE implementation, this problem did not receive attention and all allocations were filled with 0 by default for objects with constant content. To save that semantics, we engaged Z3-functionality of constant arrays, i.e. arrays with a default value. Therefore, we introduced an additional array source *ConstantWithSymbolicSize*. This source indicates, that the underlying objects are a constant array (not symbolic), but have symbolic size. Therefore, in

translation to the solver, it should receive a Z3's constant array with a default value specified in that source.

4.3 Improved lazy initialization

In Section 2 we described previously existing implementation of the lazy initialization mechanism within our fork of KLEE. In that implementation, we were forced to add additional constraints to restrict overlappings of lazily initialized memory object with any other objects. Once we added symcretes functionality, we may apply that technique to lazy initialization. The usage scheme is quite similar to the objects of symbolic size, but for now, we have explicitly defined symbolic address. Moreover, we can also use extensions with objects of symbolic size to lazily initialize memory objects as we do not know the exact size of the object, which we are dereferencing at the moment of lazy initialization. Thus, it turns out, that to lazily initialize a memory object all we need is to create a new object with symbolic size and add an equality constraint between the symcrete address and address, which have been used for dereferencing.

5. Evaluation

5.1 Experiment

For evaluation of the described features, we have used the test sets from TestComp-2022 competition [11]. Our main goal was to test the proposed approach implemented on top of the KLEE (KLEE-SYM) and make a comparison with the version of KLEE extended with lazy initialization (KLEE-LI).

We have used KLEE-LI based on the KLEE of version 2.3 with Z3 of version 4.12.1 as SMT-solver [12].

We have selected 5 different test sets with over 2000 tests per each — MemSafety-Arrays (MS-A), MemSafety-Heap (MS-H), MemSafety-LinkedLists (MS-LL), ReachSafety-Arrays (RS-A) and Termination-MainHeap (T-MH). Comparison has been made by the following metrics: instruction coverage (icov), branch coverage percentage (bcov), and numbers of found vulnerabilities (errs). Coverage has been measured with gcov [13] util.

Experiments were conducted on a workstation with CPU AMD Ryzen 7 3800X 8-Core with 16 gigabytes of RAM under the control of Linux. Execution of each test was bounded with 30 seconds timeout. As Z3 may receive complex queries, its execution time also has been bounded with 5 seconds timeout to prevent memory and time issues.

5.2 Results

Average results for tests in each source set are presented in *Table 1*.

We can notice significant improvements at ReachSafety-Arrays and MemSafety-Arrays for all parameters. These test cases used dynamic allocations of blocks with indeterminate sizes and therefore received much better results in contrast with KLEE-LI. In addition, the amount of found vulnerabilities also increased since it became possible to explore more paths that had been beyond the abilities of the engine before.

Nonetheless, we did not receive full coverage of these two test sets. One of the reasons that symbolic execution is sensible to strategies of path selection: these strategies navigate the engine through the exponential branching space. For presented test sets, the problems may come from constructions of a form presented in *Listing 6*.

Our goal is to cover the `return 0` statement. But to do that KLEE-LI should get information, that this line is reachable only if 256 is a factor of `n`. As it cannot infer such information, it will

brute force all possible variants on `n` until it will be able to reach the selected line of code. For larger programs, it may take a while to reach such statements.

On the other hand, we might see a slight deterioration in the instruction coverage and the number of errors detected on the Termination-MainHeap test set. This issue is connected to the imprecision of modelling the allocated buffer’s contents: while in reality the memory of allocated buffers is guaranteed to be initialized, KLEE models the newly allocated buffers as filled with some fixed concrete value.

Also, we’ve collected additional statistics about verdicts for the generated tests (see *Table 2*). We’ve calculated the number of generated tests for each source set (column overall), the number of execution paths that have been halted because of the inability of the old version to maintain objects of symbolic size correctly (halted), and the number of solver errors happened during program exploration, e.g. timeouts, internal errors, etc. (serrs).

Table 1. TestComp benchmarks average results

TestSet	KLEE-LI			KLEE-SYM		
	icov	bcov	errs	icov	bcov	errs
MS-A	71.8%	57.2%	346	79.5%	67.5%	680
RS-A	57.4%	45.0%	393	69.3%	61.5%	532
T-MH	91.2%	78.8%	317	90.1%	80.9%	215
MS-H	45.2%	46.2%	51	45.2%	45.7%	52
MS-LL	33.0%	30.2%	55	33.0%	30.2%	55

Table 2. Tests generated for TestComp benchmarks

TestSet	KLEE-LI			KLEE-SYM		
	overall	halted	serrs	overall	halted	serrs
MS-A	801	455	0	681	0	1
RS-A	649	238	18	539	0	7
T-MH	539	222	0	216	0	1
MS-H	58	7	0	52	0	0
MS-LL	55	0	0	55	0	0

This table demonstrates that our approach has reduced the number of internal errors in KLEE and increased the amount of non-halted branches. For the last two test sets, we did not receive any improvements in instruction and branch coverage (*Table 1*). However, for the test set MemSafety-Heap number of errors, that we classified as halted, decreased to 0. For the test set MemSafety-LinkedList, we’ve received identical results. The low percentage of coverage for these test sets is explained by a significant number of syntactically unreachable code in tested programs.

```
unsigned n <- symbolic;
char * s = (char *) malloc(n);
for (int i = 0; i < n; i++) {
    s [i] = i % 256;
}
if (s [n - 1] == 255) {
    return 0;
}
return 1;
```

Listing 6. Allocation and cycle

6. Related works

Symbolic execution with symcrete variables is an already known approach. For instance, the authors of "Deferred Concretization in Symbolic Execution via Fuzzing" [7] describe a similar approach, using symcretes to better approximate external calls with fuzzer (yet another application of symcretes). Similar to symcrete variables ideas are also used in well-known techniques of *symcretic* [14] and *concolic* [15] execution. The idea behind these methods is to combine a symbolic and concrete execution to improve performance and increase code coverage in comparison with plain symbolic execution. Unlike execution with symcrete variables, these approaches use concrete values to guide an execution, while we use symcrete variables to increase the accuracy of symbolic execution analysis.

However, the memory model can be improved without a symcrete variables approach. For instance, authors of "A bounded symbolic-size model for symbolic execution" [16] propose an approach for memory modelling, where all constraints restricting memory objects overlapping are added explicitly. To solve a problem with excessive memory consumption the authors specify a bound on size for objects with symbolic sizes. On the one hand, such a way of modelling objects with symbolic size does not require additional queries to the solver to minimize object sizes, as memory consumption becomes the responsibility of the users. On the other hand, that bound may affect the completeness of a symbolic execution engine, i.e. restrict an engine from exploring possibly reachable paths, as in some cases user will have to guess the bound to achieve higher coverage. Therefore, memory consumption will increase and performance degrade.

Another possible implementation of objects with the symbolic size is presented in the work "Symbolic-size memory allocation support for Klee" [17]. It introduces a segmented memory layout approach for KLEE symbolic execution engine. The core difference is that this work proposes a memory model, where each memory allocation lies in its memory segment. In contrast, our implementation of objects with symbolic sizes does not significantly change the memory model of vanilla KLEE, and therefore still can be considered flattened. To resolve a problem with excessive memory consumption, the authors use the same methods as described in this article: size minimization to reduce overall memory consumption and sparse structures to keep only useful data for symbolic arrays.

7. Conclusions

Accurate modelling of specific code constructions with logical constraints might be too complicated (recall the problem with external calls). We can make under or overapproximations to at least continue analysis, but with a significant loss of precision. To get things slightly better we apply the technique of symcrete variables – symbolic variables paired with concrete values for it, fitting the current constraint set.

We have proposed our implementation of dynamically recomputed symcrete values in KLEE for LLVM-programs analysis. For that, we have also enhanced the execution engine with the validity cores. Then we have shown how to engage this feature to model objects with symbolic size. To optimize the memory consumption problem, we have implemented a size minimization algorithm for objects with symbolic size and sparse storage to store only the affected solution bytes. Also, we have improved the existing mechanism of lazy initialization by address symcretization and interpretation of initialized object size as symbolic. We've also presented an implementation of this approach on top of KLEE and showed its effectiveness on several tests of Test-Comp competition.

Symcretes infrastructure is a powerful foundation for other improvements. For instance, we may use a similar approach to approximate the behaviour of external or undefined functions with fuzzers, as described in "Deferred Concretization in Symbolic Execution via Fuzzing" [7]. The

return value and function arguments, in this case, should be marked as symcretes, and calls to that function generate concrete values for symcretes.

Another interesting idea is to use a symcrete infrastructure with a type system. This might be useful if we want to test a program, which operates with polymorphic objects. Types of such objects may be considered symbolic, and therefore we have uncertainty in calls to virtual functions and sizes of underlying objects. This uncertainty can be resolved with symcretes, as it seems that we can model such behaviours with objects with symbolic sizes and calls to undefined functions.

References

Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: Communications of the ACM 56.2 (2013), pp. 82–90.

Clark Barrett and Cesare Tinelli. “Satisfiability modulo theories”. In: Handbook of model checking. Springer, 2018, pp. 305–343.

Leonardo de Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. 2008, pp. 337–340.

Haniel Barbosa et al. “cvc5: A versatile and industrial-strength SMT solver”. In: Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I. Springer. 2022, pp. 415–442.

Aina Niemetz and Mathias Preiner. “Bitwuzla at the SMT-COMP 2020”. In: arXiv preprint arXiv:2006.01621 (2020).

Corina S Pa˘sa˘reanu, Neha Rungta, and Willem Visser. “Symbolic execution with mixed concrete-symbolic solving”. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. 2011, pp. 34–44.

Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. “Deferred concretization in symbolic execution via fuzzing”. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019, pp. 228–238.

Misonijnik A. et al. “Automated testing of LLVM programs with complex input data structures”. In: Proceedings of ISP RAS 34.4 (2022), pp. 49–62.

Sarfraz Khurshid, Corina S Pa˘sa˘reanu, and Willem Visser. “Generalized symbolic execution for model checking and testing”. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer. 2003, pp. 553–568.

Cristian Cadar and Daniel Dunbar. KLEE. Version 2.3. 2022. URL: https://github.com/klee/klee/tree/v2.3.

Dirk Beyer. “Advances in Automatic Software Testing: Test-Comp 2022.” In: FASE. 2022, pp. 321–335.

Leonardo de Moura and Nikolaj Bjørner. Z3 4.12.1. Version 4.12.1. 2023. URL: https://github.com/Z3Prover/ z3/releases/tag/z3-4.12.1.

Brian Gough and Richard M Stallman. “An Introduction to GCC for the GNU Compilers gcc and g++”. In: Network Theory Ltd 258 (2004).

Peter Dinges and Gul Agha. “Targeted test input generation using symbolic-concrete backward execution”. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. 2014, pp. 31–36.

Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A concolic unit testing engine for C”. In: ACM SIGSOFT Software Engineering Notes 30.5 (2005), pp. 263–272.

David Trabish, Shachar Itzhaky, and Noam Rinetzky. “A bounded symbolic-size model for symbolic execution”. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021, pp. 1190–1201.

Michael Sima´˘cek. “Symbolic-size memory allocation support for Klee”. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2018.

Информация об авторах / Information about authors

Сергей Антонович МОРОЗОВ – студент 3-го курса Национального исследовательского университета “Высшая школа экономики”. Сфера научных интересов: методы анализа программ и оптимизации символьного исполнения.

Sergey Antonovich MOROZOV – Higher School of Economics, third-year student. Research interests: methods of program analysis and symbolic execution optimization.

Александр Владимирович МИСОНИЖНИК – старший инженер-программист компании IT Solutions Inc. Сфера научных интересов: методы эффективного поиска недостижимых состояний в символьном анализе программ.

Aleksandr Vladimirovich MISONIZHNIK – IT Solutions Inc., senior software engineer. Research interests: efficient pruning of unreachable states in symbolic program analysis.

Дмитрий Александрович МОРДВИНОВ – кандидат физико-математических наук, доцент кафедры системного программирования Санкт-Петербургского государственного университета. Сфера научных интересов: формальная верификация, синтез программ и решение систем дизъюнктов Хорна.

Dmitry Aleksandrovich MORDVINOV – PhD in Physics and Mathematics, Associate Professor at the Department of System Programming of St. Petersburg State University (SPbSU), Research interests: formal verification, program synthesis, and constraint Horn clause solving.

Дмитрий Владимирович КОЗНОВ – доктор технических наук, профессор кафедры системного программирования Санкт-Петербургского государственного университета, Сфера научных интересов: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение.

Dmitry Vladimirovich KOZNOV – D. Sc., Associate Professor, Professor St.Petersburg State University (SPbSU). Research interests: software engineering, model-driven software development, program data, machine learning.

Дмитрий Аркадьевич ИВАНОВ – начальник департамента исследований и разработок инструментальных средств компании Huawei Technologies Co., Ltd. Сфера научных интересов: инструменты разработки, анализ кода, символьное исполнение, интегрированные среды разработки.

Dmitry Arkadevich IVANOV – Huawei Technologies Co., Ltd, Director of R&D Toolchain department. Research interests: Developer Tools, Code Analysis, Symbolic execution, IDE.

DOI: 10.15514/ISPRAS-2023-35(3)-8

REDoS Detection in “Domino” Regular Expressions by Ambiguity Analysis

[1] *Antonina Nepeivoda*, ORCID: *0000-0003-3949-2164 <a_nevod@mail.ru>*
[2] *Yulia Belikova*, ORCID: *0009-0007-7829-1249, <ju.belikova@gmail.com>*
[2] *Kirill Shevchenko*, ORCID: *0009-0007-2868-153X <k.sh3vch3nko@yandex.ru>*
[2] *Mikhail Teriukha*, ORCID: *0009-0005-2825-8171 <misha37a999@yandex.ru>*
[2] *Danila Knyazihin*, ORCID: *0009-0009-6343-6809 <dak151449@gmail.com>*
[2] *Aleksandr Delman*, ORCID: *0009-0009-6885-8429 <adelman2112@gmail.com>*
[2] *Anna Terentyeva*, ORCID: *0009-0006-8547-3959 <mathhyyn@gmail.com>*

[1] *Aylamazyan Program Systems Institute of the Russian Academy of Sciences, 4a, Petra I st., Veskovo, Pereslavsky District, Yaroslavl Oblast, 152024, Russia.*
[2] *Bauman Moscow State Technical University, 5, 2-nd Baumanskaya, Moscow, 105005, Russia.*

Abstract: The Regular Expression Denial of Service (REDoS) problem refers to a time explosion caused by the high computational complexity of matching a string against a regex pattern. This issue is prevalent in popular regex engines, such as PYTHON, JAVASCRIPT, and C++. In this paper, we examine several existing open-source tools for detecting REDoS and identify a class of regexes that can create REDoS situations in popular regex engines but are not detected by these tools. To address this gap, we propose a new approach based on ambiguity analysis, which combines a strong star-normal form test with an analysis of the transformation monoids of Glushkov automata orbits. Our experiments demonstrate that our implementation outperforms the existing tools on regexes with polynomial matching complexity and complex subexpression overlap structures.

Keywords: regular expressions; ambiguity; REDoS; Glushkov automaton; transformation monoid; strong star-normal form.

For citation: Nepeivoda A.N., Belikova Yu.A., Shevchenko K.K., Teriukha M.R., Knyazihin D.P., Delman A.D., Terentyeva A.S. REDoS Detection in “Domino” Regular Expressions by Ambiguity Analysis. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 109–124. DOI: 10.15514/ISPRAS–2023–35(3)–8

Acknowledgements: The first author was partially supported by Russian Academy of Sciences, research project No. 122012700089-0.

Выявление REDoS ситуаций в регулярных выражениях структуры «домино»

[1] *А.Н. Непейвода,* ORCID: *0000-0003-3949-2164 <a_nevod@mail.ru>*,
[2] *Ю.А. Беликова,* ORCID: 0009-0007-7829-1249 *<ju.belikova@gmail.com>*
[2] *К.К. Шевченко,* ORCID: *0009-0007-2868-153X <k.sh3vch3nko@yandex.ru>*
[2] *М.Р. Терюха,* ORCID: *0009-0005-2825-8171 <misha37a999@yandex.ru>*
[2] *Д.П. Князихин,* ORCID: *0009-0009-6343-6809 <dak151449@gmail.com>*
[2] *А.Д. Дельман,* ORCID: *0009-0009-6885-8429 <adelman2112@gmail.com>*
[2] *А.С. Терентьева,* ORCID: *0009-0006-8547-3959 <mathhyyn@gmail.com>*

[1] *Институт программных систем РАН им. А.К. Айламазяна, Россия, 152024, Ярославская обл., с. Веськово, ул. Петра I, д. 4а.*
[2] *Московский государственный технический университет имени Н.Э. Баумана, 105005, Россия, Москва, ул. Бауманская 2-я, д. 5/1.*

Аннотация. Ситуация отказа в обслуживания регулярных выражений (REDoS) возникает в случае высокой вычислительной сложности сопоставления строки с выражением и встречается во многих библиотеках регулярных выражений таких языков, как PYTHON, JAVASCRIPT, C++. В данной статье рассматривается класс регулярных выражений, которые создают угрозу возникновения REDoS, однако не распознаются как уязвимые рядом существующих программных систем. Предлагается производить оценку степени неоднозначности таких выражений посредством комбинирования проверки на строгую звёздную нормальную форму и анализа трансформационного моноида автомата Глушкова, построенного по входному регулярному выражению. Эксперименты показывают, что данный подход оказывается эффективен при оценке полиномиальных неоднозначностей в регулярных выражениях со сложной структурой перекрытий.

Ключевые слова: регулярные выражения; неоднозначность; REDoS; автомат Глушкова; трансформационный моноид; сильная звёздная нормальная форма.

Для цитирования: Непейвода А.Н., Беликова Ю.А., Шевченко, К.К. Терюха М.Р., Князихин Д.П., Дельман А.Д., Терентьева А.С. Выявление REDoS ситуаций в регулярных выражениях структуры «домино». Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 109–124 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–8

Благодарности. Первый автор осуществлял работу над проектом при частичной поддержке Российской Академии Наук, номер НИР 122012700089-0.

1. Introduction

Popular regular expression (regex) engines typically use non-deterministic finite automata (NFA) as their internal representation for regexes. This choice is motivated by the flexibility of the NFA concept, which can be extended to support a wider range of regex operations with little effort. For instance, back-references and lookaheads can be easily added to the NFA model. Although, in theory, every string can be matched against a regex in linear time using deterministic finite automata (DFA) conversion, popular regex engines may admit exponential matching time due to a phenomenon called "catastrophic backtracking".

This phenomenon occurs only for a specific class of regular expressions. For example, consider the regex `(a|b)*a`, which is non-deterministic due to the unavoidable non-determinism in the transition to the last occurrence of the letter `a`. However, every string has a unique parsing tree with respect to this regex. In contrast, the regex `(a*b*)*` has an infinite number of accepting parsing trees for any given string, as inner Kleene stars can degenerate to the empty word, causing a combinatorial explosion of parse paths. Intuitively, the latter regex can be considered "bad", while the former is considered "good".

Matching against "bad" regexes can yield a situation called a Regular Expression Denial of Service (REDoS), when the matching time grows super-linearly and can cause performance issues in, for

instance, a web service that uses such a regex to parse user input. To avoid these situations, it is essential to detect unsafe regexes and replace them with safe equivalents.

The number of research papers mentioning the REDoS problem has increased rapidly in the last decade [1]–[7]. Several tools have been developed to detect REDoS, using both static analysis and random search. Some of these tools aim to detect the entire class of extended regexes, while others focus on academic ones. However, for a class of simple regexes, which are not safe in theory, the tools considered either take too long time to process, or give an incorrect answer, falsely witnessing their safety. These regexes usually have overlapping, but not completely coinciding, structure of the expressions under the Kleene stars (being a simple analogue of dominoes in the Post Correspondence Problem). An example of such a regex is $(baa|ab)^*(b|\varepsilon)(a(ba|a)ba^*b)^*(aab)^*$: the ambiguity occurs both in prefixes $(baa)^n$ and $(ab)^n$, which can be constructed in several ways from primitive "dominoes".

Thus, the two natural research questions arise:

- do the "domino" regexes really contain REDoS situations w.r.t. the modern regex engines?
- if the answer is yes, what methods can deal with such regexes in order to analyse them without blow-up of the analysis time because of the overlaps?

The main contributions of the paper are:

- a method for REDoS situations detection, utilizing properties of non-deterministic finite automata and their transition monoids. This approach is novel, since previous static-analysis-based methods use NFA intersection. For "domino" regexes our method is shown to perform better than the open-source analogues REGEX STATIC ANALYZER [3], RESCUE [5], and REVEALER [2].
- experimental testing of the relevance of the NFA model used and the vulnerabilities found, by investigating real regex engines behaviour on the attack strings.

The method is implemented only for the academic regexes for now. Surprisingly, for this case, the tested open-source tools perform significantly worse on domino tests, especially for polynomial REDoS situations.

The paper is organized as follows. Section 2 contains preliminaries on finite automata, and theoretical concepts that are used further. The proposed REDoS detection method is given in Section 3, preceded by lemmas used for its optimisation. Section 4 discusses relevance of the chosen model with respect to the real regex matching engines, and provides a result of comparative testing of our method and three other open-source REDoS detection tools. We discuss the results of the experiments and the related works in more detail in Section 5. Section 6 concludes the paper.

2. Preliminaries

We denote automata with calligraphic $\mathcal{A}$; states are denoted with the letters q and Q, or with the set of these letters (if an automaton is a result of a closure operation). The empty word is denoted by ε; concrete elements from the input alphabet are denoted with a, b, c, …, and letter parameters are denoted with γ; ω and η denote word parameters. We use only the basic academic regular expression constructing operations: concatenation (which is omitted in notation), alternation (denoted with |), and Kleene star (denoted with *). If r is a regex, $\mathcal{L}(r)$ denotes its language.

Let us recall basic definitions and describe the finite automata models used in this paper.

2.1 Finite Automata

Definition 1. *A non-deterministic finite automaton (NFA) is a tuple <S, Σ, q_0, F, δ>, where:*

- *S is a state set;*

- Σ *is a terminal alphabet;*
- δ *is a set of transitions of the form* $<q_i, (\gamma_i \mid \varepsilon), M_i>$, *where* $q_i \in S$, $\gamma_i \in \Sigma$, $M_i \in 2^S$;
- $q_0 \in S$ *is the initial state;*
- $F \subseteq S$, *is a set of final states.*

Every transition in an NFA maps a pair $<q_i, (\gamma_i \mid \varepsilon)>$ into a set of states, contrary to transitions in a deterministic finite automaton (DFA), which map every pair (q_i, γ_i) (where γ_i is essentially not equal to ε) to a single state. Thus, if a word is parsed by a DFA, the parse trace is always unique (i.e., DFAs are unambiguous); in an NFA, there can be a set of parse traces for a single word. This set can even be infinite in case of NFA with ε-transitions. The notation $q_i \to^{\gamma} \ldots$ is overloaded to denote either NFA transition $<q_i, \gamma, M_i>$ (written as $q_i \to^{\gamma} M_i$) or a transition to a single state belonging to M_i (written as $q_i \to^{\gamma} q_j$). Existence of a path from q_i to q_j marked by ω in Σ^* is also denoted by $q_i \to^{\omega} q_j$.

An NFA can be transformed into an equivalent DFA using a textbook subset-constructing algorithm `Determinize`, which generates states of the DFA corresponding to the sets of the states of the initial NFA resulted in the transitions along the same input symbols.

The NFA models used in regex engines are primarily based on the classical Thompson construction, which provides an algorithm for transforming a regex into an NFA that recognizes the same language. While the implementation details of the transformation may vary, the experiments presented in Section 4 provide evidence that the Thompson model remains relevant for identifying inefficient regexes with respect to NFA-based parsing engines.

In the following descriptions, we only give details of the constructed NFAs in terms of their states and transitions, without mentioning the alphabet construction.

Definition 2. *Thompson NFA (denoted with* `Thompson(r)`*) is constructed from a regex* `r` *as follows. At any construction step except processing concatenations, the new initial state* q_r *and the new final state* Q_r *are introduced, and the transition set is updated depending on the regex operation.*

- *Every single letter* γ *generates a primitive automaton with the only transition* $q_\gamma \to^{\gamma} \{Q_\gamma\}$.
- *If* $\mathcal{A}_1$ = `Thompson(`r_1`)`, $\mathcal{A}_2$ =`Thompson(`r_2`)`, *and* q_i *and* Q_i *are their initial and final states, respectively, then* `Thompson(`$r_1 \mid r_2$`)` *is constructed by merging the* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *states sets and transitions sets, and introducing the transitions* $q_{alt} \to^{\varepsilon} \{q_1, q_2\}$; $Q_1 \to^{\varepsilon} \{Q_{alt}\}$; $Q_2 \to^{\varepsilon} \{Q_{alt}\}$.
- `Thompson(`$r_1 r_2$`)` *is again constructed by merging* `Thompson(`r_i`)` *states and transitions sets, and making* q_1 *the initial state,* Q_2 *the final state, with the additional transition* $Q_1 \to^{\varepsilon} \{q_2\}$.
- `Thompson(`r_1^*`)` *is constructed introducing transitions* $q_* \to^{\varepsilon} \{q_1, Q_*\}$, $Q_1 \to^{\varepsilon} \{q_1, Q_*\}$.

The Thompson construction algorithm ensures that any NFA produced by the algorithm has a unique final state and that each state has at most two outgoing and two incoming transition arcs. The uniqueness of the final state implies that the reverse NFA for `Thompson(`r`)` is exactly `Thompson(`r^R`)`, where r^R is the reverse of the regex r. Additionally, all subregex automata can be treated as isolated directed acyclic graphs, which makes the construction easily extensible and decomposable. An example of a Thompson automaton for a regex is shown in *Fig. 1*. The states labels follow the corresponding regex operations given in Definition 2.

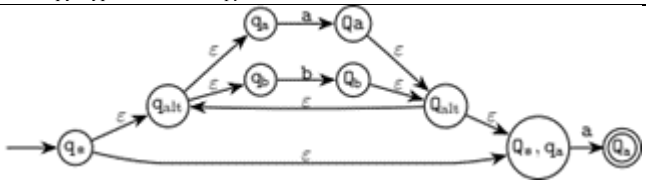


Fig. 1. Thompson automaton for `(a|b)*a`

One drawback of the Thompson construction is that it introduces non-deterministic transitions corresponding to alternating operations (i.e., alternatives or Kleene stars), even in the cases when the regex itself imposes no non-determinism (e.g. for the regex `a(a|b)*`, which is a reverse of the regex shown in *Fig. 1*). To avoid the redundant non-determinism, the regex engine RE2 [8] processes such strongly deterministic regexes (also known as 1-unambiguous regexes [9]) constructing another NFA based on the regex structure, but without ε-transitions. This automaton is known as the Glushkov automaton since 1960s, and in the last two decades it attracted considerable interest, shown to be efficient and extensible to construct deterministic parsing engines for a larger class of regexes (such as memory finite automata for the regexes with back-references [10]). The Glushkov automaton is shown in *Fig. 2*.

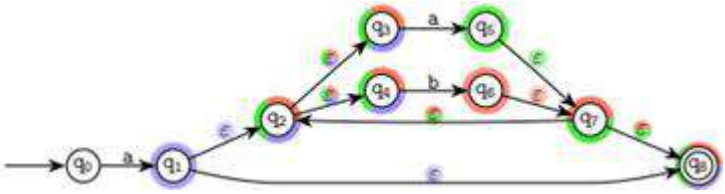


Fig. 2a. `Thompson(a(a|b)*)` *with colored* ε*-closures*

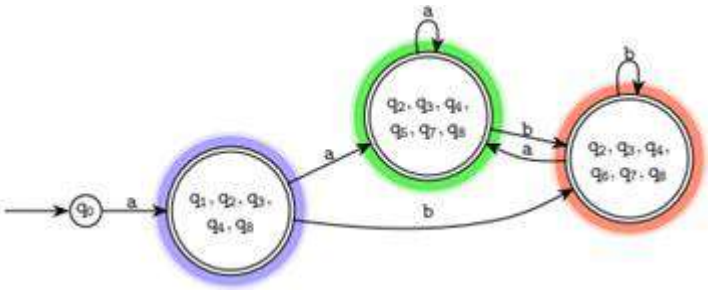


Figure 2b. `Glushkov(a(a|b)*)`

The classical Glushkov construction is based on so-called follow-relation on linearised regexes. By construction, every state in the Glushkov automaton except the initial state corresponds to an occurrence of some γ in Σ in the input regex r; conversely, any letter occurrence in the regex r corresponds to exactly one state in `Glushkov(`r`)`, whose incoming arcs are all marked with γ. Now we can reformulate this property in the terms of Thompson and Glushkov automata.

Proposition 1. *There is a bijection from state set in* `Glushkov(`r`)` *minus the initial state to state set* Q_γ *in* `Thompson(`r`)` *(where* Q_γ *are final states of the primitive automata reading* γ*).*

In the paper [11], it was shown that `Glushkov(`r`)` could also be obtained from `Thompson(`r`)` merging its ε-closures.

Definition 3. *Given an NFA* $\mathcal{A}$ *and its state q,* ε*-closure of q is the maximal set of states reachable from q following only* ε*-transitions.*

Closure-merging[1] *ε-free automaton (denoted with* `RemEps(`$\mathcal{A}$`)`*) is constructed from* $\mathcal{A}$ *as follows:*

- *its states are ε-closures of the states of* $\mathcal{A}$*;*
- *if state* q_1 *belongs to closure* C_i*, state* q_2 *belongs to* C_j*, and there is a transition* $q_1 \to^{\gamma} \{..., q_2, ...\}$ $(\gamma \neq \varepsilon)$ *in* $\mathcal{A}$*, then there is a transition* $C_i \to \{..., C_j, ...\}$ *in* `RemEps(`$\mathcal{A}$`)`*.*

An example of closure-merging operation is given in *Fig. 1* and *Fig. 2*, the nodes belonging to a closure are highlighted with the same color.

2.2 Transformation Monoid of NFA

Let us consider an automaton with no useless states and ε-transitions. Its transitions over the alphabet Σ and the states set 2^Q form the function $F : \Sigma \times S \to 2^S$ taking a pair $<\gamma, q_i>$. This function, when curried and specialized in the first argument, becomes F_γ: $S \to 2^S$ (where $\gamma \in \Sigma$). We can form a monoid over the set of such partially specialized functions (transformations) if we continue them on strings as follows: $F_{\omega 1} \circ F_{\omega 2} = F_{\omega 2 \omega 1}$. Then associativity is provided "for free", given associativity of string concatenation, and ε becomes the monoid unit, because $F_\omega \circ F_\varepsilon = F_{\omega\varepsilon} = F_\omega = F_{\varepsilon\omega} = F_\varepsilon \circ F_\omega$ holds. The state transformations are denoted by the corresponding strings ω.
The formal definition is as follows [12].

Definition 4. *Given an ε-free automaton* $\mathcal{A}$ *over the alphabet* Σ*, its transformation monoid* $\mathcal{M}$ = `TransMonoid(`$\mathcal{A}$`)` *is the monoid of transformations imposed by elements of* Σ^* *on the states of* $\mathcal{A}$*.*

The monoid construction does not depend on the choice of the final or initial states of $\mathcal{A}$ (except the condition that all the states are useful, i.e. reachable and producing), thus, instead of classical NFAs, the monoid is based on a labelled transition system. Since the set of functions $S \to 2^S$ is finite, the transformation monoid of an NFA always contains a finite number of equivalence classes. The pair $<M, R>$, where M is a finite set of lexicographically minimal elements of the equivalence classes and R is a set of simplification rules is considered a standard representation of the transformation monoid. Such a representation for `TransMonoid(Glushkov(a(a|b)*))` is given in *Fig. 3a*, *Fig. 3b*, and *Table 1*. The monoid representation uncovers some useful NFA properties. For example, we can immediately conclude that the words `aa` and `ab` are synchronizing, since for all q_i, $q_i \to^{aa} q_2$, $q_i \to^{ab} q_3$, and no other transition is possible.

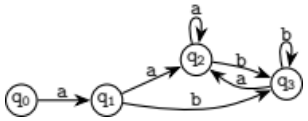


Fig. 3a. Labelled transition system of NFA

```
bb → b        aaa → aa
aab → ab      aba → aa
baa → ba      bab → bb
```

Fig. 3b. Rewriting rules of NFA

[1] This ε-removal construction differs from the standard textbook ε-removal algorithm, since it changes states, and not only transitions. This strategy allows the algorithm to succeed in conversion from Thompson to Glushkov.

Table 1. Equivalence classes of NFA

	q_0	q_1	q_2	q_3
a	$\{q_1\}$	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
b	{ }	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
aa	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
ab	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
ba	{ }	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$

2.3. Ambiguity of NFAs and REDoS

Intuitively, the worst-case scenario for backtracking-based matching of a string against a regex r occurs when the matched string has a prefix η_1 with a large set of parse paths, and a suffix η_2 such that $\eta_1\eta_2 \notin \mathcal{L}(r)$. In this case, in order to determine that $\eta_1\eta_2$ is not recognizable by r, a regex engine must backtrack through all the parse variants of η_1. Obviously, we can choose such a suffix η_3 that $\eta_1\eta_3 \in \mathcal{L}(r)$, and $\eta_1\eta_3$ will still have a large number of parse trees (although the regex engine will report a success after finding a first one).

Therefore, worst-case matching time depends on the upper bound on the parse paths in a regex.

In the domain of finite automata, the following definition is used [13], [14].

Definition 5. *A degree of ambiguity of an NFA $\mathcal{A}$ is a worst-case bound on the number of paths recognizing an input string (in a length of the string).*

The ambiguity of NFAs is known to be either a constant, an exponential, or a polynomial [13]. If the ambiguity degree of $\mathcal{A}$ is non-constant, it is said $\mathcal{A}$ has an infinite degree of ambiguity (IDA). A standard acronym for exponential ambiguity degree is EDA.

A minimal EDA-generating regex example is $(a|a)^*$. A minimal example of a regex producing IDA but not EDA automaton is a^*a^*. For regexes such that $(a^*b^*)^*$, $\texttt{Glushkov}(r)$ is unambiguous, despite $\texttt{Thompson}(r)$ is EDA. We can notice that in $\texttt{Thompson}((a^*b^*)^*)$, a special situation occurs: there is a loop inside an ε-closure of a state (i.e., there is at least one Kleene star in a regex iterating over an expression r_E s.t. $\varepsilon \in \mathcal{L}(r_E)$). Further we show that such a case is one of the few possible exceptions when $\texttt{Thompson}(r)$ and $\texttt{Glushkov}(r)$ have distinct ambiguity degrees.

The following criterion estimates the degree of ambiguity in any NFA.

Theorem 1. • NFA $\mathcal{A}$ *satisfies IDA condition iff there exist states q_1, q_2 in $\mathcal{A}$, and a word ω such that $\mathcal{A}$ contains paths from q_1 and q_2 to themselves, and a path from q_1 to q_2 all accepting the word ω.*

- NFA $\mathcal{A}$ *satisfies EDA condition iff there exists a state q_1 in $\mathcal{A}$, and a word ω such that $\mathcal{A}$ contains two distinct loops from q_1 to itself both accepting the word ω.*

We can also say than if EDA occurs in an NFA, then $\exists\ q_i, q_j, q_k$, where q_j and q_k are distinct, such that there exist words ω_1 and ω_2 such that both q_k and q_j are reachable from q_i following a path reading the word ω_1, and q_i is reachable from both q_k and q_j following a path reading the word ω_2 (see *Fig. 4*).

After the work [9], we use the term "orbit of state q" for the maximal strongly connected component containing q. We assume that orbits are non-trivial, i.e. contain at least one transition. If a state q of $\mathcal{A}$ satisfies EDA criterion for some ω, then all states belonging to its orbit also satisfy EDA. Thus, to check the EDA condition, it is sufficient to check if any state of some strongly connected

component of an NFA satisfies EDA; for the IDA condition, it is sufficient to check if there are two strongly connected components satisfying it.

Fig. 4a. The EDA situation

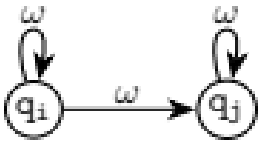


Fig. 4b. The IDA situation

An approach to the IDA and EDA detection used in the REDoS analysers [3], [4] tests the above criterion constructing single or double intersections of automaton $\mathcal{A}$ with itself. Although the intersection construction can be done in polynomial time on an NFA size, it may lead to large NFAs if there are many crossing components (i.e., matching the same string sets) in the initial NFA.
The IDA criterion can be also reformulated in the terms of transformation monoids.

Proposition 2. *An ε-free automaton $\mathcal{A}$ satisfies IDA iff its transformation monoid contains an equivalence class ω such that for some states q_i, q_j, $q_i \in F_\omega(q_i)$, $q_j \in F_\omega(q_j)$, and $q_j \in F_\omega(q_i)$.*

Using this criterion for an initial NFA "as is" is highly impractical: if the NFA contains non-crossing components, the transformation monoid becomes exponentially huge. However, with some refinements, we observed that the monoid criterion can be applied (and even be fast) in the cases when the intersection criterion is slow. Moreover, Proposition 2 provides explicit construction of a string with the ambiguity, allowing the analysing algorithm to reconstruct the REDoS situation easily. First, take any NFA path from the initial state of $\mathcal{A}$ to q_i, recognizing some prefix η_1. Then pump ω to construct an infix with superlinear number of parse trees, and then take some string η_2 such that any path from q_j recognizing η_2 does not end in a final state of $\mathcal{A}$. The string $\eta_1\omega^n\eta_2$ will force an NFA parsing device to do superlinear backtracking.
If the monoid criterion is applied to the orbit automaton of state q, the REDoS pump can be constructed as well. Just choose some η_1, η_2 such that $q_0 \to^{\eta_1} q$, and $\forall q_F \in F$ in the condition $q \to^{\eta_2} q_F$ is not satisfied.

3. Our approach

As a starting point, we prefer to use the Thompson automaton as a preliminary NFA model for a regex since regex matching engines rely on it in their internal algorithms, and experiments in Section 4 demonstrate that the Thompson construction is suitable for analysing real REDoS. However, in order to apply the monoid criterion, we must first eliminate ε-transitions in the regex and ensure that the removal of ε-transitions does not affect the degree of ambiguity.

Definition 6. *A regular expression r is said to be in a star-normal form (SNF) if for each its subexpression $(r')^* \varepsilon \notin \mathcal{L}(r')$.*

Let us say that r *is in the strong star-normal form (SSNF) if it is in SNF and none of its subexpressions* $(r')^*$ *contains an alternation* $r_1 | r_2$*, where* $\varepsilon \in \mathcal{L}(r_1)$ *&* $\varepsilon \in \mathcal{L}(r_2)$*.*

The following theorem is the main theoretical result of the paper.

Theorem 2. *If* r *is SSNF, and* Thompson(r) *is infinitely ambiguous, then* Glushkov(r) *is also infinitely ambiguous. Moreover, the overall ambiguity degrees (exponential or polynomial) of* Thompson(r) *and* Glushkov(r) *coincide.*

In order to prove Theorem 2, we use the statement proven in [11] mentioned above: RemEps(Thompson(r)) = Glushkov(r), where RemEps is the transformation described in Section 2, and the following auxiliary proposition.

Proposition 3. *If* a_1 *and* a_2 *are distinct letter occurrences in* r*, and* q_{A1}*, and* q_{A2} *are final states of the elementary Thompson automata for* a_1 *and* a_2*, then* q_{A1} *and* q_{A2} *never belong to a single* ε*-closure of a state in* Thompson(r)*.*

Proof of Proposition 3:

Every final state of the elementary automaton for a single letter has a unique ingoing edge, marked with the given letter. No other construction adds ingoing edges to the final states of the subautomata used in the construction. Thus, the states q_{A1} cannot be reached from q_{A2} along ε-transitions, and vice versa.

Proposition 3 allows us to construct the bijection between final states of the elementary subautomata of Thompson(r) and all the states except the initial one in Glushkov(r), mentioned in Section 2.

Proof of Theorem 2:

Let r be in the strong star-normal form. All the strongly connected components of Thompson(r) and Glushkov(r) correspond to expressions under Kleene stars.

If some strongly connected component of Thompson(r) contains an EDA, then there exists a state q, two distinct states q_1 и q_2 and $\alpha_1, \alpha_2 \in \Sigma \cup \{\varepsilon\}$, words ω_1, ω_2 such that $<q, \alpha_1, q_1> \in \delta$, $<q, \alpha_2, q_2> \in \delta$, $q_1 \rightarrow^{\omega 1} q$, $q_2 \rightarrow^{\omega 2} q$, satisfying $\alpha_1\omega_1 = \alpha_2\omega_2$. Let us denote the path from q to itself following through q_1 by P_1, and the similar path following through q_2 by P_2.

If $\alpha_1\omega_1 = \varepsilon$, then there is an ε-loop from q to itself, which contradicts the SSNF condition. Thus, we can take the first letter of $\alpha_1\omega_1$ belonging to Σ, say a. Let us consider the final states q_1', q_2' of elementary Thompson automata for a in the paths P_1 и P_2.

If q_1' is not equal to q_2', then their ε-closures are also distinct, which implies the EDA situation in Glushkov(r).

Let q_1' and q_2' coincide. We recall that we chose the states q, q_1, q_2, such that the first edges in the paths P_1 and P_2 outgoing from q (and ingoing in q_1 and q_2), are distinct.

Let $q_1'=q_1$ (or $q_1'=q_2$). The state q_1' has a single ingoing edge, namely the one outgoing from q and marked with a. But $q_1'=q_2'$, and q, being a predecessor of q_2' in the path P_2, must occur in its initial fragment twice, thus, there is a path from q to q recognizing ε. This contradicts the SSNF condition.

Let q_1' to be distinct both from q_1 and q_2, but to coincide with q_2'. Let us again consider the ingoing edge in q_1' marked with a. Let this edge to outgo from some state q_0. Then there are the two distinct paths from q into q_0 reading the empty word, again contradicting SSNF. Thus, EDA in the Thompson automaton leads to EDA in the Glushkov automaton.

Now let Thompson(r) contain IDA, but not an EDA. Then r contains the distinct subexpressions r_1 and r_2 under the Kleene stars, both recognizing a same word ω, such that the states corresponding to r_1 are not reachable from the states generated by r_2. Thus, r_1 and r_2 contain the same letter a with occurrences in the positions i and j, $i<j$, and the state for a_i in Glushkov(r) is not reachable from the state for a_j. Then Glushkov(r) contains an IDA, which is not an EDA. This completes the proof.

Thus, it is sufficient to test r for the strong star-normal form property and then, if necessary, continue the ambiguity analysis operating with the Glushkov automaton, having significantly less states. If there are loops in ε-closures, the further analysis is not needed: these loops already produce EDA situations.

Given a state q in $\mathcal{A}$ and its orbit M, an orbit automaton of q is automaton M_q including all states and transitions from M, having q as is the initial state, and whose final states are either final states of $\mathcal{A}$ or states with outgoing transitions outside the orbit M in $\mathcal{A}$.

If we choose one state q_i from each strongly connected component C_i of $\mathcal{A}$, then testing an IDA criterion for `TransMonoid`(M_{qi}) is enough to reveal all EDA situations.

However, in the case of a polynomial IDA, we must test pairs of the strongly connected components (together with the transitions from one component to another) and building a monoid for any such pair-generated NFA is too time- consuming. Thus, we use the following simple necessary condition for the polynomial IDA.

Proposition 4. *Let C_1, C_2 be distinct strongly connected components of $\mathcal{A}$. If $\mathcal{A}$ contains a polynomial IDA within the components, then there exist two states, $q_1 \in C_1$, $q_2 \in C_2$, such that* `Determinize`*($\mathcal{A}$) contains a subset state including both q_1* and *q_2. Moreover, such a subset state occurs also in* `Determinize(Reverse(`$\mathcal{A}$`))`.

Although the determinization algorithm is exponentially hard in the worst case, it is known to be fast in most practical cases [16]. Thus, the subset test accelerates candidates search for the polynomial IDA. However, it is not sufficient, which can be shown by analysing regex `(a|b)*(b|c)(a|c)*` whose Thompson automaton contains no IDA.

The pseudocode of the complete algorithm[2] is given in *Fig. 5*. There $\mathcal{A}_{q1+q2}$ includes the orbit automata M_{q1} and M_{q2} of q_1 and q_2, and all states reachable from M_{q1} and reaching M_{q2} together with their transitions. Its initial state coincides with initial state of M_{q1}, and its final states are final states of M_{q2} (ignoring final states of $\mathcal{A}$ belonging either to M_{q1} or to the intermediate states). The condition "c_1 `reaches` c_2" ensures that the component c_2 is reachable from c_1, and they do not coincide. Operator c[1] takes a first state from the component c (since the `Ambiguity.TransMonoid` and determinization tests results do not depend on the choice of the initial state in the orbit automata[3]). Function `SCC`($\mathcal{A}$) returns all strongly connected components of $\mathcal{A}$.

4. Experiments

4.1 Data Set

In order to evaluate the effectiveness of our approach on the "domino" regexes, a dataset of 100 academic regexes was generated. The regexes satisfy the following properties:

- their length and alphabet are small (not more than 50 terms and not more than 5 distinct letters);
- they have iterated elements;
- all are in SSNF.

The first condition allows significant subexpression languages overlap, without blowing up the regex length. However, the test set contains not only complex dominoes, but also regexes with simple ambiguity situations like `b*c(ac|(aa|a)*d)*`. The second condition is necessary for

[2] The trial implementation of the method is given on https://github.com/bmstu-iu9/Chipo-Kleene/tree/ambiguity.

[3] Absence of any useless states is guaranteed, because all the states are reachable from each other.

REDoS situations. The third condition mostly excludes the trivial SSNF test, returning EDA value using our method too quickly.

```
if ¬ SSNF(r) then
  return EDA
  A ← Glushkov(r)
end if
C ← SCC(A)
for c ∈ C do
  q0 ← c[1]
  if Ambiguity(TransMonoid(M_{q0})) then
    return EDA
  end if
end for
for c1, c2 ∈ C do
  if c1 reaches c2 then
    q1 ← c1[1]
    q2 ← c2[1]
    if SubsetPairs(Determinize(A_{q1+q2}))
    ∩ SubsetPairs(Determinize(Reverse(A_{q1+q2}))) ≠ ∅
    then
      if Ambiguity(TransMonoid(A_{q1+q2})) then
        return IDA
      end if
    end if
  end if
end for
return Safe
```

Fig. 5. The overall algorithm schema

We explored the dependence of the regexes matching time from the input length on the popular engines in PYTHON, JAVASCRIPT, C++, JAVA 8, JAVA 11, GO, and RUST.

In order to detect super-linear dependencies, it is necessary to generate potentially attacking input, for which the string pumping method is used. The attacking input must match a pattern of the three components: a prefix that satisfies the regular expression, a pumping core whose repetition can lead to a rapid increase in the number of parsing paths (i.e., malicious pump), and a suffix whose mismatch leads to catastrophic backtracking.

The results obtained by applying JAVASCRIPT, PYTHON, C++ and JAVA 8 standard regex engines are the same, according to them, the data set contains 34 exponential, 36 polynomial and 30 safe regexes. In addition, the experiments indicated that JAVA 11 standard regex engine handles some polynomial and exponential cases, but when the length of the input data increases significantly, it throws a stack overflow exception, which may be due to the introduction of the local storage of indexes to the regex module in the 11 version of JAVA.

The regexes are safe for GO and RUST engines, which are based on the deterministic structures. Nevertheless, it was noted that there are frequent single outliers in trends when matching strings in GO.

During testing, we observed that polynomial regexes only lead to critical matching times (more than 1 minute) with significant input string lengths (approximately more than 500 characters), while expressions that have exponential matching complexity can reach critical time when parsing even

relatively small input strings. In the simplest case, such a time explosion can be achieved with regexes that have large star nesting or multiple alternatives under a star *q*uantifier. For instance, the PYTHON, JAVASCRIPT, JAVA 8, and C++ regex engines are vulnerable to attacks in the case of the `((a*)*)*` regex, and even the optimized JAVA 11 engine, which successfully handles double star nesting, reaches critical time processing such an expression.

However, more non-trivial cases were encountered in the proposed data set. For example, the regex `b(ab((a|b(a*a)*)a*b*)*|a*aaaa*)*`, when matched against the input of 32 characters that satisfies the pattern with prefix – `b`, pump – `abab`, suffix – `bbd`, achieves the following timings: PYTHON engine – over 3 minutes, JAVA 8 – over 3 minutes, JAVA 11 – 0.80 minutes, C++ – over 3 minutes, JAVASCRIPT – 1.73 minutes.

In general, the REDoS vulnerability degree coincides with the theoretical expectations, taking into account the asymptotic growth of the ambiguity function for the corresponding Thompson automata. Non-SSNF regexes cause critical time explosion, which is evidence that the regex engines do not apply SSNF transformation to their input. In addition to non-SSNF regexes, critical REDoS situations occur on polynomial ambiguities iterated under a Kleene star.

4.2 Comparing with other tools

We evaluated the effectiveness of the proposed approach by comparing it with three state-of-the-art open-source tools for detecting vulnerabilities in regexes: RSA [3], [17], a static analysis tool, RESCUE [5], [18], a genetic fuzzing tool, REVEALER [2], [19], an automated hybrid analysis tool that uses static and dynamic approaches.

The qualitative results of the experiments are described in *Table 2*. To evaluate the effectiveness of detection of vulnerable and safe regexes, we used F_1-score, where true positive values are all vulnerable regular expressions that were classified as exponential or polynomial, the absence of results due to a timeout is taken into account as a false result, also we used the error rate, where a cumulative error on all classes of regexes – total error rate and a classification error among vulnerable regexes – vulnerable error rate. It should be noted that RESCUE does not support the exponential-polynomial classification, therefore, not all values were calculated for this tool.

Table 2. Evaluation results

Tool	F_1-score	Total error rate	Vulnerable error rate
RSA	0.90	0.13	0.00
ReScue	0.39	-	-
Revealer	0.55	0.45	0.04
Our method	1.00	0.00	0.00

The results of measuring the execution time for the considered tools are shown in Table 3. When measuring time, all extended features of the tools were disabled, and their parameters were optimized. For each class of correctly classified regexes: exponential, polynomial, safe, unsafe (union of vulnerable regexes), the average running time (μ) and the standard deviation (σ) of this value were estimated, the number of timeouts was also calculated.

Additionally, we chose 25 regexes with non-SSNF structure, which are analysed in our method by the preliminary ε-loop test. While our approach proved to be the fastest (which is not a surprise, provided the algorithm structure), the static part of REVEALER also had 100% success rate on this set, although, taking at average 4x more time.

It is important to note that the theoretical results obtained by using static analysis methods, determining ambiguity degree of the Thompson automata, completely coincide with the experimental results obtained when testing the domino regexes on the PYTHON, JAVASCRIPT, JAVA 8, and C++ regex engines. This is a strong witness that regexes declared safe by dynamic or combined methods are their false negatives.

Table 3. Time measurements

	Exp		Poly		Safe		Unsafe		Timeouts
Tool	$\mu(s)$	$\sigma(s)$	$\mu(s)$	$\sigma(s)$	$\mu(s)$	$\sigma(s)$	$\mu(s)$	$\sigma(s)$	
RSA	1.895	2.614	3.480	3.748	0.836	0.341	2.578	3.221	13
ReScue	-	-	-	-	0.940	1.724	8.803	6.263	43
Revealer	0.410	0.035	0.402	0.021	0.320	0.065	0.409	0.033	0
Our method	0.836	1.059	1.178	1.259	0.484	0.400	1.014	1.169	0

From the test results, we can conclude that the detection efficiency of the static analyser is high, but in non-trivial exponential or polynomial cases such as `(baa|ab)*b(a(b|a)ba*b*(aab)*`, timeouts occur. The recognition efficiency of ReScue and Revealer tools on this data set is low. However, the proposed approach has the maximum *q*uality of vulnerability detection, the average execution time is also superior to other implementations. This is partly explained by its narrow domain: testing only academic regexes. But RSA also aims at the academic regexes, and still has several timeouts; on the other hand, it seems that extension of REDoS-detection tools to non-academic regexes made them to miss almost all polynomial REDoS with domino structure.

5. Discussion and Related Works

Initially, our finite automata transforming tool was not designed to reveal REDoS situations. However, attempts to use open-source tools like Regex Static Analyser or ReScue to analyze simple academic regexes with non-trivial ambiguity structure failed. The main purpose of the work was educational, so we designed our algorithm in such a way that it not only detects vulnerabilities, but also demonstrates them on the automata graphs, at the cost of longer execution

time. Since the tool was initially designed for demonstrations, only core academic regexes were considered. The algorithms used in the monoid-based approach have poor worst-case complexity, so its efficiency, compared to RSA and ReScue, was a real surprise.

What features of the analysers caused such a situation? RSA uses NFA intersection construction, based on the well-known paper of Mohri et al [14]. To detect polynomial ambiguities, the algorithm requires self-intersecting an NFA twice. The automata intersection problem is known to be PSPACE-complete [20], [21], thus, every additional intersection results in a significant slowdown. Maybe that is the main cause why the polynomial detection results in timeouts in RSA. The monoid and determinization algorithms are known to be worst-case exponential. However, the determinization is proven to be fast 4 in average [16], while the monoid representation depends heavily on the automata structure and, implemented to orbit automata, generates significantly fewer e*q*uivalence classes, compared to the case when automata are not cyclic. Another well-known problem in static analysers is dealing with ε-transitions, which can ruin the intersection construction, as well as the monoid. Surprisingly, the tools do not use the simple and natural conversion to the Glushkov construction preceded by the SSNF test.

Error rate of static tools is usually much lower than in tools using genetic algorithms and fuzzing, since REDoS-provoking strings can be disguised, requiring several explicit iterations to construct, or be combined from several alternative subexpressions under an iteration. Even using two approaches in Revealer cannot help to find vulnerabilities, if the malicious pump is hidden in overlaps and crossing occurrences. For example, in paper [6], four REDoS classes are provided, based on a regex structure, and the regex `a*(ab)*a(ba)*` satisfies neither of them, because the vulnerability appears due to the crossing occurrence of the string ab on the border of the two orbits, whereas the expressions under Kleene stars have languages with empty intersection, which makes the regex “seemingly safe”. A similar pattern-based approach is used in [7], resulting in the same sort of false negatives. So, regex-based heuristics showed themselves to be too weak as compared to the model NFA analysis in the domino ambiguity cases.

If a malicious pump for a regex is found, the natural question arises: how to correct the regex? We did not consider the whole implementation of the regex correction, but implemented a trial algorithm constructing a 1-unambiguous regex, if it exists [9]. However, for most regexes with overlaps, even if the equivalent 1-unambiguous regex can be built, the algorithm given in [9] produces exponentially longer result, as compared to the input, processing all overlap combinations separately. A more optimistic regex correcting heuristic is the Star Normal Form transformation: it is performed in linear time and produces regexes approximately of the same length. Moreover, the SSNF transformation is rather local, does not require transition to NFA, and can be applied even to extended regexes, which is useful, taking in account that non-SSNF regexes cause critical REDoS w.r.t. PYTHON and JAVASCRIPT regex engines. In general, the question what theoretical results can be used to fix REDoS regexes, is still a subject of research.

6. Conclusion

The research resulted in the following answers to our research questions.

- ***RQ1:*** *how relevant is NFA static analysis w.r.t. to popular regex engines?*

  Our experiments demonstrated that the Thompson NFA model is entirely suitable for evaluating REDoS situations concerning the most widely used regex engines, including PYTHON, JAVASCRIPT, JAVA, and C++. Interestingly, although the GO regex machine uses conversion to DFA, it still produces surges on some ambiguous regexes with complex structures. The RUST DFA engine proved to be the most stable.

- ***RQ2:*** *what features of the REDoS analysers considered cause errors and time explosion on the regexes with complex overlap structure? How they can be processed reliably with less risk of time explosion?*

  We found out that considering orbit automata (instead of performing ambiguity analysis on the entire NFA) and using the Glushkov construction, preceded by the Strong Star Normal Form test, do not result in any loss of relevance, but significantly speed up the static analysis.

  Another interesting approach is to use monoid analysis as the primary ambiguity-detecting algorithm instead of NFA intersection analysis. If there are multiple substring overlaps in the orbits, this method performs significantly faster. However, if the overlaps are small, the number of equivalence classes in the monoid increases dramatically, making the intersection method more preferable.

  We also provided experimental evidence that the genetic search REDoS detection methods still miss complex REDoS cases, easily detected by static NFA analysis approaches.

Despite our approach proved itself to be efficient and reliable on the test set of domino regexes, it still requires many refinements. First, the monoid construction may explode if we take large alphabets, so the input regexes may need some alphabet factorization. E.g., if no overlaps are contained within a long string, then this string sometimes can be considered as a single letter. Second, it would be interesting to test the method on extended regexes approximation, and to combine the monoid-based and intersection-based ambiguity detection algorithms.

References

[1]. Davis J. C., Coghlan C. A., Servant F., and Lee D. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In Proc. of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018. pp. 246-256. DOI: 10.1145/3236024.3236027.

[2]. Liu Y., Zhang M., and Meng W. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In Proc. of the 2021 IEEE Symposium on Security and Privacy (SP), 2021. pp. 1468-1484. DOI: 10.1109/SP40001.2021.00062.

[3]. Van der Merwe B., Weideman N., and Berglund M. Turning evil regexes harmless. In Proc. of the South African Institute of Computer Scientists and Information Technologists, 2017. pp. 1-10. DOI: 10.1145/3129416.3129440.
[4]. Weideman N., van der Merwe B., Berglund M., Watson B. W. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In Proc. of the Implementation and Application of Automata - 21st International Conference. 2016. pp. 322-334. DOI: 10.1007/978-3-319-40946-7_27.
[5]. Shen Y., Jiang Y., Xu C., Yu P., Ma X., Lu J. ReScue: Crafting regular expression DoS attacks. In Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018. pp. 225-235. DOI: 10.1145/3238147.3238159.
[6]. Li Y., Sun Y., Xu Z., Cao J., Li Y., Li R., Chen H., Cheung S.-C., Liu Y., Xiao Y. RegexScalpel: Regular expression denial of service (ReDoS) defense by Localize-and-Fix. In Proc. of the 31st USENIX Security Symposium (USENIX Security 22), 2022. pp. 4183-4200.
[7]. Li Y., Chen Z., Cao J., Xu Z., Peng *Q*., Chen H., Chen L., Cheung S. ReDoSHunter: A combined static and dynamic approach for regular expression DoS detection. In Proc. of the 30th USENIX Security Symposium (USENIX Security 21), 2021. pp. 3847-3864.
[8]. Google. Official public repository of RE2 library. Available at: *https://github.com/google/re2*, accessed 01.07.2023.
[9]. Bruggemann-Klein A. and Wood D. One-unambiguous regular languages. Information and Computation, vol. 140, no. 2, 1998. pp. 229-253. DOI: 10.1006/inco.1997.2688.
[10]. Freydenberger D. D., Schmid M. L. Deterministic regular expressions with back-references. Journal of Computer and System Sciences, vol. 105, 2019. pp. 1-39. DOI: 10.1016/j.jcss.2019.04.001.
[11]. Allauzen C., Mohri M. A unified construction of the Glushkov, Follow, and Antimirov automata. In Proc. of the Mathematical Foundations of Computer Science, 2006. pp. 110-121. DOI: 10.1007/11821069_10.
[12]. Eric Pin J. Mathematical foundations of automata theory. Available at: https://www.irif.fr/~jep/PDF/MPRI/MPRI.pdf, accessed 01.07.2023.
[13]. Weber A., Seidl H. On the degree of ambiguity of finite automata. Theoretical Computer Science, vol. 88, no. 2, 1991. pp. 325-349. DOI: 10.1016/0304-3975(91)90381-B.
[14]. Allauzen C., Mohri M., Rastogi A. General algorithms for testing the ambiguity of finite automata. In Proc. of the Developments in Language Theory, 2008. pp. 108-120. DOI: 10.1007/978-3-540-85780-8_8.
[15]. Bruggemann-Klein A. Regular expressions into finite automata. Theoretical Computer Science, vol. 120, no. 2, 1993. pp. 197-213. DOI: 10.1016/0304-3975(93)90287-4.
[16]. Almeida M., Moreira N., Reis R. On the performance of automata minimization algorithms. In Proc. of the 4th Conference on Computability in Europe, 2008. pp. 3-14.
[17]. Weideman N. Regex static analyzer. Available at: *https://github.com/NicolaasWeideman/RegexStaticAnalysis*, accessed 01.07.2023.
[18]. Shen Y., Jiang Y., Xu C., Yu P., Ma X., Lu J. Rescue. Available at: *https://github.com/2bdenny/ReScue*, accessed 01.07.2023.
[19]. Liu Y., Zhang M., Meng W. Revealer. Available at: *https://github.com/cuhkseclab/Revealer*, accessed 01.07.2023.
[20]. Gelade W., Neven F. Succinctness of the Complement and Intersection of Regular Expressions. In Proc. of the 25th International Symposium on Theoretical Aspects of Computer Science, 2008. pp. 325-336. DOI: 10.4230/LIPIcs.STACS.2008.1354.
[21]. Birget J., Margolis S. W., Meakin J. C., Weil P. Pspace-complete problems for subgroups of free groups and inverse finite automata. Theoretical Computer Science, vol. 242, no. 1-2, 2000. pp. 247-281. DOI: 10.1016/S0304-3975(98)00225-4.

Информация об авторах / Information about authors

Антонина Николаевна НЕПЕЙВОДА – научный сотрудник Института Программных Систем РАН. Сфера научных интересов: теория формальных языков, программная семантика, математическая логика и функциональное программирование.

Antonina Nikolaevna NEPEIVODA – researcher in the Program Systems Institute of RAS. Research interests: formal language theory, program semantics, mathematical logic, and functional programming.

Юлия Андреевна БЕЛИКОВА – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: теория формальных языков, искусственный интеллект, анализ данных.

Yulia Andreevna BELIKOVA – student of the Bauman Moscow State Technical University. Research interests: formal language theory, artificial intelligence, data analysis.

Кирилл Константинович ШЕВЧЕНКО – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: анализ данных и машинное обучение.

Kirill Konstantinovich SHEVCHENKO – student of the Bauman Moscow State Technical University. Research interests: data science and machine learning.

Михаил Романович ТЕРЮХА – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: интернет вещей и распределённые вычисления.

Mikhail Romanovich TERIUKHA – student of the Bauman Moscow State Technical University. Research interests: internet of things and distributed systems.

Данила Павлович КНЯЗИХИН – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: абстрактная алгебра.

Danila Pavlovich KNYAZIHIN – student of the Bauman Moscow State Technical University. Research interests: abstract algebra.

Александр Дмитриевич ДЕЛЬМАН – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: конструирование компиляторов и облачные вычисления.

Aleksandr Dmitrievich DELMAN – student of the Bauman Moscow State Technical University. Research interests: compiler design and cloud computing.

Анна Сергеевна ТЕРЕНТЬЕВА – студент Московского государственного технического университета им. Н. Э. Баумана. Сфера научных интересов: конструирование компиляторов.

Anna Sergeevna TERENTYEVA – student of the Bauman Moscow State Technical University. Research interests: compiler design and optimisation.

DOI: 10.15514/ISPRAS-2023-35(3)-9

Alias Analysis and Calculus based on Segmentation Address Memory Model

I.A. Parfenov, ORCID: 0009-0004-2889-0380 <*parfenov_2001@mail.ru*>
Innopolis University
1, Universitetskaya Str., Innopolis, 420500, Russia

Abstract. We present a straightforward implementation of a simplified imperative programming language with direct memory access and address arithmetic, and a simple static analyzer for memory leaks. Our study continues a line of research attempted (in Innopolis University in years 2016-2022) on alias calculi for imperative programming languages with decidable pointer arithmetic but differs by memory address model – we study segmented memory model instead linear one.

Keywords: Imperative programming; memory address model; memory safety; memory leaks; static analysis

For citation: Parfenov I.A. Alias Analysis and Calculus based on Segmentation Address Memory Model. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 125-136. DOI: 10.15514/ISPRAS-2023-35(3)-9.

Анализ и исчисление алиасов, основанное на сегментированной модели адресации памяти

И.А. Парфенов, ORCID: 0009-0004-2889-0380 <*parfenov_2001@mail.ru*>
Университет Иннополис,
Россия, 420500, Татарстан Республика, г. Казань, г. Иннополис, ул. Университетская, д. 1

Аннотация. Мы представляем простую реализацию упрощенного императивного языка программирования с прямым доступом к памяти и адресной арифметикой, а также простой статический анализатор утечек памяти. Наше исследование продолжает линию исследований, предпринятых (в Университете Иннополис в 2016-2022 годах) по исчислению алиасов для императивных языков программирования с разрешимой арифметикой указателей, но отличается моделью адресации памяти — мы изучаем сегментированную модель памяти вместо линейной.

Ключевые слова: императивное программирование; модель адресации памяти; безопасность памяти; утечки памяти; статический анализ.

Для цитирования: Парфенов И.А. Анализ и исчисление алиасов, основанное на сегментированной модели адресации памяти. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 125–136 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–9.

1. Introduction

There are various different instruments for program code development. One of the areas that has to be improved for programming languages is the safety and correctness of successfully compiled programs. The C programming language, like some others, has pointers and direct memory access, which is a powerful and, at the same time, uncontrollable instrument, whose safety depends only on

the programmer. Those programming languages need some validation techniques for checking the safety and correctness of using those features.

The alias calculus is the mathematical model which operates on an abstract, simplified programming language with dynamic memory (accessible by explicit or implicit pointers) and some rules, using which it can validate if the program is memory safe. This theory can be expanded to real programming languages.

In this paper, firstly, a new variant of the alias calculus is suggested and studied up to some extent. Then, based on this theory, we present a compiler for a simple model C-like programming language with direct memory access (via pointers). The compiler has been implemented from scratches. For this programming language, a set of programs has been written, and some metrics and statistics of their executions have been collected and studied. Finally, we present a static validator for memory leak safety for programs (with pointers) written in our model language. We hope that this instrument can be successfully used in real programming.

2. Literature review

2.1 Anderson's Model

Andersen's pointer analysis model [1] is the most closest to alias calculus among commonly used static analysis models. However, it describes a little different, more simplified, pointer-to model. Nevertheless, definitions and properties introduced there are necessary for this scope's analysis. Roughly speaking, Andersen' pointer analysis is based on theory of equality for uninterpreted functional symbols. The algorithm traverses the program by statements and calculates for every pointer set of other pointers, to which it can be equal. Such a set is called points-to set. During traversing, once an assignment is met, a constraint "point-to set of sources is subset of point-to set of destination" is created. After the constraints are collected, they are solved. The content of the work is overcomplicated, though.

2.2 Alias Calculus

A simplified description of alias calculus and some other information out of this scope is described in [2]. Informally, alias relation is a structure, which specifies for every variable, to which pointer variable does it belong. The cited paper presents a set of simple operations: assignment, allocation and deallocation, if-statements and loops, which affect the alias relation. The purpose of calculus rules is static over approximation Q of actual alias relation after execution of a program S for a given alias relation P before the program execution, i.e., such relation Q that Hoare triple $[P]S[Q]$ to be true. For example, assignment statement copies aliases to destination replaced by source and removes aliases, which contain source; if-statements calculate relations for all branches and unite them.

The algorithm from [2] was implemented in the Eiffel Verification Environment and can be used through the AutoProof module. The approach used in the algorithm, presented in our paper, is, however, different and will be explained in detail in the corresponding section. One of the main differences is the memory model used: in [2] memory consists of abstract addresses while in our model, for every state and for every variable, alias relation describes the meta-identifier of allocated space and shift relative to the meta-identifier. This allows swapping allocated space for variables without triggering the validator.

2.3 Separation Logic

Separation logic [3] and [4] is an extension of the first-order logic for specification of the programs over dynamic memory (heap) in Hoare assertions $[P]S[Q]$. It operates on a heap, which is addressed, using a "separating conjunction" operator, which checks if objects hold different parts of the heap. There were proposed ways to handle unrestricted memory access with not only static arrays, but also

dynamic arrays and recursive functions. The concept of separation logic is widely used in different proof assistants and frameworks; hence, it can also be used for validating programs in this scope. Separation logic semantics is based on a model comprising stores (to represent static memory) and heaps (to represent dynamic memory), which are finite-domain maps from variables' identifiers and, respectively, locations (accessible via pointers or addresses which are particular numbers), to data values (e.g. integers). There are two major heap models in use: linear or flat (where each location is capable of storing simple data values), and segmented (where locations can store compound data like arrays with static size).

2.4 MoRe Language

[5] presents and describes the MoRe language, which allows more flexible actions on pointers' addresses in comparison to Andersen's one. The cited paper describes the target theory in the clearest and most understandable way, so this was the starting point for our research. MoRe language presents the linear address arithmetic and has a separate stack and heap address spaces. The language has direct memory access and address arithmetic; hence its memory model fully represents C programming language address memory model. There are only integer and pointer data types in MoRe. The algorithm traverses the program and calculates a set of configurations at every moment. The configuration consists of three objects: a set of address variables, a set of address expressions and a set of pairs of "synonyms" – variables, which point to one cell in current configuration. For recalculation the state an operator "aft" was introduced, which for every possible state and statement properly defines a new state after execution of the statement. The syntax grammar of this language is given in *Fig.* 1. Bachelor Thesis [6] presents simple implementation and analysis of MoRe language. Bachelor Thesis [7] implements simplified C language with MoRe language interface, which can be compiled using LLVM. The syntax and semantics of this thesis' implemented language is close to MoRe's.

$$
\begin{aligned}
P ::= \quad & skip \mid var\ V = C \mid V := T \mid V ::= cons(C*) \mid \\
& \mid [V] := V \mid V := [V] \mid dispose(V) \mid (P\,;P) \mid \\
& \mid (if\ then\ P\ else\ P) \mid (while\ do\ P)
\end{aligned}
$$

Fig. 1. The syntax grammar of MoRe language: start variable is P, C is constant integer, and C is list of integers with "," character between them*

3. Methodology

In this section we informally introduce and overview a simplified model language Alias. Though the real implemented language has same syntax as presented here, it's semantic is developed more practically oriented and proposes new instruments.

3.1 Alias Programming Language Overview

The implemented version of the Alias language has/offers

- Two types – integer and pointer (to be tracked in analysis)
- Variable definitions, assignments, and annotations (assumptions)
- Blocks, If- and While-statements, Procedures.

Program may be split on multiple files. BNF syntax definition of language is given in *Fig. 2*. However, the semantics are very restricted.

- *Type* ::= int | ptr
- *Program* ::= *Block*
- *Block* ::= {[*Statement*]}
- *Statement* ::=
  - *Block* — Block
  - *def Ident Type* — Definition
  - *Ident := Expression* — Assignment
  - *Ident <- Expression* — Movement
  - *free(Ident)* — Deallocation operator
  - *if (Expression) Block* — If statement
  - *if (Expression) Block else Block* — If/Else statement
  - *while (Expression) Block* — While statement
  - *func([int Ident | ptr Ident Integer Integer] Block* — Function definition
  - *call Ident([Ident])* — Function call
- Expression ::=
  - *Ident*
  - *Integer*
  - *$Ident* — Dereference
  - *Expression + Expression*
  - *alloc(Int)* — Allocation operator

Fig. 2. The syntax grammar of implemented language

3.2 Outlines of validation algorithm (static semantics)

Memory safety validation is done using the following method (algorithm).

- Program (text) is parsed line by line maintaining (in form of states) a set of known relations "pointer points to cell in heap" but ignoring any information about integer variables.
  - At some stages the states (known relations "pointer points to cell in heap") can split, as there is no information about integer variables.
  - If in a current line there is no pointer variable, which points to any heap cell, then it means memory leak happened.
  - if there is a dereference of a pointer variable, which at some state points out of allocated area, then access violation happened.

3.3 Configuration

Every configuration contains

I: A set of local variables/identifiers, which have pointer type.

A: A set of allocated cells in the heap (each cell in the form "Meta-variable + Integer-phase")

S: For every identifier in I appointed cell in A, or an exceptional value "OUT".

3.4 Legal Types of Assignments

There are three types of assignments:

1) int := int – i.e., an integer expression is assigned to an integer variable

2) [ptr -> ptr] := ptr – i.e., a pointer expression is assigned to a pointer

3) [ptr -> int] := int – i.e., an indirect assignment to integer variable

Only the second type effects on configuration.

Note that storage pointer variables on heap doesn't effect on configuration. Hence validation of multidimensional arrays of structures, for example, is not supported by our analysis.

3.5 Some optimizing assumptions

We make the following (informal) assumptions about programs (for boosting of validator).

- The number of local variables is not very big, while the number of heap cells can be very big, but (as now) is assumed constant.
- Since the number of configurations grows exponentially, we implement 'assume' statement, which filters the configurations which pass given condition (but programmer is responsible for the correctness of this assumption).
- The current number of configurations is counted, so the programmer can get number of configurations in real time in IDE.

3.6 Static semantics for pointers

Program traversed recursively. For the following statements corresponding actions made:

- Block affects only on visibility scopes of variables. It doesn't change state.
- Definition affects only on visibility scopes of variables. It doesn't change state.
- Assignment depending on types does following:
  - Destination has pointer type, and source has pointer type. If assignment has form $a := b + x$ and in some configuration $a = a_v + a_p$ and $b = b_v + b_p$ then in new state this configuration has $a = b_v + (b_p + x)$. For example, if there was configuration with ($b = \_0+3$) and statement $a := b + (-1)$ was executed, the next configuration will be with ($b = \_0 + 2$). If after this in some configuration there is no a_v, then memory leak happened.
  - Destination has pointer type, and source has integer type. For every configuration and every allocated cell new configuration created where destination points to such cell.
  - If destination has integer type, the state is not changed.
- Assumption works as a guard, i.e., it removes configurations, where the assumption condition is false. If assumption has form $assume(a = b + x)$ and in some configuration $a = a_v + a_p$ and $b = b_v + b_p$, then if $a_v \neq b_v$ or $a_p \neq b_p + x$, then condition is false. If assumption has form $assume(a < b + x)$ and in some configuration $a = a_v + a_p$ and $b = b_v + b_p$, then if $a_v \neq b_v$ or $a_p \geq b_p + x$, then condition is false.
- If-statement is traversed in the following steps. Firstly, the first branch is validated. Then the sizes of all lists, which were allocated during this are saved and set to 0. Finally, the second branch is validated, and finally the sizes of lists are restored. *If there is allocation in one branch, then the list will be added to states, but it won't appear in any configuration in second branch, hence it is guaranteed, that an alert will be shown*. (Probably it is a solvable problem, we can force to allocate to the variable the same size at the last assignment in both branches.)
- While-statement is traversed in the following steps. The body is validated, and if state has been changed, the body validated again. There is a threshold (set in validator) for number of these iterations, after exceeding which, it is assumed that the loop is infinite. The variables declared in a loop are scoped in the loop.

- A function actually is a procedure, its definition contains set of formal parameters (as arguments) in its signature. Each pointer parameter has two associated integer values, which guarantee the minimal number of sells before and after a call. The function doesn't return values, but can change its pointer arguments (i.e., actual arguments are passed name to function).
- Function call contains parameters as actual arguments of function. If in some configuration some pointer variable (passed to the function as an actual argument) doesn't satisfy the minimal size of allocated space, then it causes a run-time error.

4. Implementation

This section describes implemented language, which is based on model language described previously, but mostly oriented on practical usage.

4.1 Overview

By default, the whole process of building and execution consists of the following sequential stages.

- Parsing `calias` parses input files and builds abstract syntax tree;
- Validation `calias` traverses tree and does static analysis;
- Compilation `calias` traverses tree and writes equivalent x86 assembly code;
- Assembly `nasm` builds object file;
- Linking `gcc` links object file and provides its `malloc` and `free` functions.

4.2 Tool-chain for the Alias Language

The compiler is implemented using language *C++* for *GNU G++* compiler and preferably uses *C++17* standard. The implementation can be found in corresponding GitHub repository. The output executable is called `calias`.

For front-end no lexical and parser generating tools, or a framework for development of domain specific languages were used, both lexer and syntax parser were implemented from scratches.

4.3 Validation

The validation is done as traversing the abstract syntax tree with passing and modification a *context*.

A context consists of the following components (though its implementation is a bit more complicated):

- stack of variables;
- stack of functions;
- vector of sizes of packets;
- set of states.

A state is a vector, which for every declared variable contains

- either the pair consisting of a packet, in which it lays, and a phase (i.e. a shift relative to the beginning of packet, to which the variable points);
- or a special value `OUT`.

Note, that here we use a terminology that differs from terminology in the section 3: context here is used instead of state, and state here is used instead of configuration (since this terminology is commonly adopted in program languages implementation community).

4.4 Rules definitions

This is a formal description of working process of validator. It omits some non-important cases, for more clear understanding.

The rule is described in two lines. Conclusions are written in the bottom line $A \vdash B \to C$ and premises – in the top line $D \vdash E \to F$. This means, that if we have to traverse node *B* of abstract syntax tree and the current context is *A*, then we have to create new context *D*, do recursive call on node *E*, which will return context *F* and then return context *C*. If a rule has no premises, it is an axiom (i.e., no further recursive calls).

Let us introduce some notation conventions. Meta-variable *V S* stands for variable stack, *FS* – for function stack, *PS* – for vector of packet sizes, and *SS* – for set of states. If the actual value of some of the listed meta-variables does not change in a rule, then it is presented implicitly, while any change of actual value must be specified in the rule explicitly. For example, if there is a line $C[FS] \vdash statement \to C[FS : foo]$, then it means that the output context is almost the same as input, but the value of *FS* (to which *foo* is appended to the end of the function stack).

Operation ":" appends the value to the end of the stack; it is also used to denote, that the element has instances in the structure. Operation "::" concatenates two stacks or vectors; it is also used to denote, that the elements of second list are presented in the first list (neglecting the order). As usual, (x, y) stays for a pair of two elements and $x := y$ denotes an update assigning the value of *y* to variable *x*.

There are following additional operators:

- *packet(x, S)* returns the identifier of the packet, to which the variable *x* is bound in state *S*;
- *phase(x, S)* returns the phase with respect to the beginning of the packet, to which variable *x* is bounded in state *S*;
- *value(x, S)* returns a pair consisting of *packet(x, S)* and *phase(x, S)*;
- *packet_size(x)* returns the size of packet *x* (remark that it is unique in all states).

The CHECK operator works as a guard, i.e., it is used to evaluate the expression (after CHECK), and if it is false, stops validation with corresponding error message.

4.5 List of Rules

1) Block: Remember the size of stack of variables. Traverse all statements in body, and crop stack of variables to previous size.

$$\frac{C \vdash S_1 \to C_1;\ \ldots;\ C_{n-1} \vdash S_n \to C_n}{C[V S, FS] \vdash \{S_1, \ldots S_n\} \to C_n[VS, FS]}$$

2) Definition: Append the variable name to the stack of variables.

$$\frac{}{C[VS] \vdash \text{def a type} \to C_2[VS : a]}$$

3) Assignment: Different behavior for integer and pointer types.

For integer we just need to check the right part is a valid expression.

$$\frac{C[VS : a] \vdash \text{expr} \to C}{C \vdash \text{a} := \text{expr} \to C}$$

There are three options for assignments with pointers – alloc, shift by a constant, and more complicated expressions in the right-hand side.

Alloc expression creates an additional packet.

$$\frac{}{C[VS : a, PS, SS] \vdash \text{a} := \text{alloc (x)} \to C[[PS : x], \forall S \in SS \to value(a, S) := (size(PS), 0)]}$$

Shift (i.e., pointer + constant integer) assigns the corresponding value.

$$\frac{}{C[VS :: [a,b], SS] \vdash \text{a := b + x} \rightarrow \\ C[\forall S \in SS \rightarrow value(a,S) := (packet(b,S), phase(b,S) + x)]}$$

For all other cases the state into states, where the variable points to one of all possible allocated cells, and check right part expression.

$$\frac{C \vdash \text{expr} \rightarrow C}{C[VS : a, SS] \vdash \text{a := expr} \rightarrow \\ C[\forall S \in SS \rightarrow \forall packet\, p, x \in [0, packet\ \ size(p)) \rightarrow value(a,S) := (p,x)]}$$

4) Movement: Check, that pointer at left size if correct, and check the right part expression.

$$\frac{C \vdash expr \rightarrow C \; C \vdash \text{a <- expr} \rightarrow C}{\text{CHECK} \forall S \in SS\, phase(a,S) \in [0, packet\ \ size(packet(a)))}$$

5) Free: Check, that the pointer has phase zero, and have same packet in all states. Assign packets of all pointers, which point to this packet, to *OUT*.

$$\frac{}{C[PS] \vdash \text{free(a)} \rightarrow \\ C[PS \rightarrow packet(p) := 0,\ \forall S \in SS \rightarrow \forall x, packet(x,S) = p \rightarrow value(x,S) := (OUT, 0)] \\ \text{CHECK} \forall S \in SS : packet(a,S) = p \, and \, phase(a,S) = 0}$$

6) Function definition: Flush all variables and append argument variables. Each pointer variable which has nonzero pre size lays in own packet with size equal to pre size. Check body. At the end check that all pointer variables lays in different packets with at least post size distance from end of packet and have same packet in all states. Restore variables and append function.

$$\frac{[[a,b],[foo],[in_a, in_b], a := (a,0), b := (b,0)] \vdash block \rightarrow C_2}{C[FS] \vdash \\ func\ foo\ (def\ a\ ptr\ in\ a\ out\ a,\ def\ b\ ptr\ in\ b\ out\ b)\ block) \rightarrow C[FS : foo] \\ \text{CHECK} \forall var\, x, S \in SS\, packet(x,S) = \\ x \, and \, phase(x,S) \in [0, packet\ size(x) - \\ out_x) \, and \forall var\, x \neq var\, y \, packet(x,S) \neq packet(y,S)}$$

7) Function call: Check, that all pre conditions are satisfied: all argument variable lay in different packets with at least pre size distance from end of packet and have same packet in all states. Remove all passed packets, as if they were freed, and create new packet for each argument variable.

$$\frac{}{C[FS : foo, PS, SS] \vdash call\, foo(args) \rightarrow C[PS :: [out_a, out_b], \\ \forall S \in SS \rightarrow value(a,S) := (new\, a, 0), b := (new\, b, 0)] \\ \text{CHECK} \forall x \in args\ S \in SS\ phase(x,S) \in [0, packet\ size(x) - in_x) \, and \forall x \neq y \in args \\ packet(x,S) \neq packet(y,S)}$$

4.6 Compilation

The compilation is done into *Intel x86 Assembly*. The compiler using almost same structure as validator. But its context is adapted for compilation. The compilation is done as traversing abstract syntax tree and building assembly code, which is the assembled using *nasm* and linked using *gcc*, which provides implementations of functions *malloc* and *free*.

4.7 Assembly structure

There are rules for compilation, which are defined the same way, as for validation. Though, they are not interested in scope of this thesis.

- There is an enter point of the program;
- There is declarations of functions *malloc* and *free*, their implementations have to be provided;
- The System V ABI [8] is used, which makes this file compatible with programs written is *C* language;
- The 32-bit assembly is used, thus the only data types have same size of four bytes;
- Only the simple general-purpose instructions are used;
- The expressions push calculated result on the top of current stack. The binary operators do recursive call of one operand, then pushes stack and do recursive call of the other operand;
- There are no optimizations.

4.8 The IDE

The IDE is implemented from scratches in language *C++* for *GNU G++* compiler and preferably uses *C++17* standard. It widely uses *NCurses* library for implementation text editor. The implementation can be found in corresponding GitHub repository. The output executable is called `ideal`.

5. Evaluation: examples of memory errors

5.1 Detected Errors with one Configuration

In this section will be presented examples of programs (each with a simple error) that have only one configuration on every state.

```
def a ptr
a := alloc (3)
a := alloc (2)
```

Listing 1. Example of memory leak

After the second assignment, there is a configuration (this is the only one configuration in this state), where there is no page, which was allocated first. The validator will show corresponding error on the third line.

```
def a ptr
a := alloc (3)
a := a + 4
def b int
b := $a
```

Listing 2. Example of access violation while dereference

In the fifth statement there an attempt to dereference the pointer, while there is a configuration, where this pointer points out of page (this is the only one configuration in this state). The validator will show corresponding error on the fifth line.

```
def a ptr
a <- 4
```

Listing 3. Example of access violation while movement

In the second statement there is an attempt to move value by pointer, while there is a configuration, where this pointer out of page (this is the only one configuration in this state). The validator will show corresponding error on the second line.

```
def a ptr
a := alloc (3)
a := a + 1
free (a)
```

Listing 4. Example of access violation while free

In the fourth statement there is an attempt to free page by pointer, while there is a configuration, where this pointer is not at the beginning of page (this is the only one configuration in this state). The validator will show corresponding error on the fourth line.

5.2 Detected Errors with multiple Configurations

Next let us discuss examples of programs which have multiple configurations in every state.

```
def a ptr
if (1) {
    a := alloc (3)
}
```

Listing 5. Example of memory leak on branching

In the end of body of if statement there is memory leak, since there is a configuration, where there is no allocated page (there are two configurations: with if and without). In general, the allocations can only be places at root blocks in function bodies.

```
def a ptr
def b ptr
a := alloc (3)
b := alloc (4)
def c ptr
if (1) {
    c := a + 0
}
else {
    c := b + 0
}
free (c)
```

Listing 6. Example of unpredictable free

In the `free` statement there are two configurations, but in these configurations the variable `c` points to different pages. It is restricted, as there is no way to continue validation.

6. Conclusion

In this work in progress paper, firstly we briefly review some approaches to memory safety analysis. Then we proceed to a new variant of alias calculus and propose several changes, stemmed from the *C* programming language memory model. Finally, we describe our implementation of a model language, our static analysis tool, and present several experiments showing analysis' potential (as we believe).

Still, we need to try validator on a "large" source code file containing more than 100 lines of code. Right now, we foresee a problem with scaling our analysis to "large" programs and on programs in a programming language from the real world. Additionally, a crucial missing piece in the theory is the handling of dynamic arrays and recursive functions.

References

[1]. L. O. Andersen, "Program analysis and specialization for c programming language," in DICU, [Online]. Available: http://www.cs.cornell.edu/courses/cs711/2005fa/papers/ andersen-thesis94.pdf, May 1994.

[2]. S. V. A. Kogtenkov B. Meyer, "Alias calculus, change calculus and frame inference," in Science of Computer Programming, [Online]. Available: http://is.ifmo.ru/articles_en/2013/meyer-calculus-2013.pdf, Nov. 2013.

[3]. J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in Carnegie Mellon University, [Online]. Available: https://www.cs.cmu.edu/~jcr/seplogic.pdf, Jul. 2022.

[4]. P. O'Hearn, "Communications of the ACM" in Carnegie Mellon University, [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3211968, Feb. 2019.

[5]. A. V. N.V. Shilov A. Satekbayeva, "Alias calculus for a simple imperative language with decidable pointer arithmetic," in Novosibirsk Computing Center, [Online]. Available: https://nccbulletin.ru/files/article/shilov_satekbayeva_vorontsov.pdf, 2014.

[6]. L. I. Lygin, "Alias calculus in C-like languages," 2021.

[7]. G. Dolgov, "Implementing alias calculus for c programming language using llvm," 2022.

[8]. A. J. Michael Matz Jan Hubicka, System V application binary interface, [Online]. Available: https://refspecs.linuxbase.org/elf/x86 64-abi-0.99.pdf, Jul. 2012.

Информация об авторах / Information about authors

Игорь Андреевич ПАРФЕНОВ – бакалавр в области информатики и вычислительной техники университета Иннополис. Сфера научных интересов: низкоуровневое и системное программирование.

Igor Andreevich PARFENOV – Bachelor of Informatics and Computer Engineering. Research interests: low level and system programming.

DOI: 10.15514/ISPRAS-2023-35(3)-10

Application of Design Patterns in the Development of the Architecture of Monitoring Systems

A.A. Pasynkova, ORCID: 0009-0006-4842-1105 <*aapasynkova1@yandex.ru*>
O.L. Vikentyeva, ORCID: 0000-0002-8991-4719 <*ovikenteva@hse.ru*>
HSE University, 38 Studencheskaya str., Perm, 614070 Russian Federation

Abstract. This article explores the relevance of using design patterns in the development of the architecture of monitoring systems. The increasing complexity of modern monitoring systems has made it challenging to maintain and evolve them. The use of design patterns can address these challenges by providing reusable solutions to common problems in monitoring system architecture. This article reviews the literature on monitoring systems and design patterns and identifies appropriate design patterns for monitoring system architecture. The article also analysis the requirements for monitoring systems and demonstrates how design patterns can be used to meet these requirements. The results show that the use of design patterns can improve the maintainability, flexibility, reliability, compatibility and scalability of monitoring systems. This article is relevant to software architects, developers, and system administrators who are involved in the development and maintenance of monitoring systems.

Keywords: design patterns; monitoring systems; architecture; monitoring system requirements.

For citation: Pasynkova A.A., Vikentyeva O.L. Application of design patterns in the development of the architecture of monitoring systems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 137-150. DOI: 10.15514/ISPRAS-2023-35(3)-10

Проектирование архитектуры системы мониторинга на основе паттернов проектирования

А.А. Пасынкова, ORCID: 0009-0006-4842-1105 <*aapasynkova1@yandex.ru*>
О.Л. Викентьева, ORCID: 0000-0002-8991-4719 <*ovikenteva@hse.ru*>
Национальный исследовательский университет ВШЭ,
614070, Россия, г. Пермь, ул. Студенческая, д. 38.

Аннотация. В данной статье исследуется актуальность использования шаблонов проектирования при разработке архитектуры систем мониторинга. Возрастающая сложность современных систем мониторинга усложняет их обслуживание и эволюцию. Использование шаблонов проектирования может решить эти проблемы, предоставляя многократно используемые решения для распространенных проблем в архитектуре систем мониторинга. В этой статье дается обзор литературы по системам мониторинга и шаблонам проектирования и определяются подходящие шаблоны проектирования для архитектуры систем мониторинга. В статье также анализируются требования к системам мониторинга и демонстрируется, как можно использовать шаблоны проектирования для удовлетворения этих требований. Результаты показывают, что использование шаблонов проектирования может улучшить удобство обслуживания, гибкость, надежность, совместимость и масштабируемость систем мониторинга. Эта статья предназначена для архитекторов программного обеспечения, разработчиков и системных администраторов, которые занимаются разработкой и обслуживанием систем мониторинга.

Ключевые слова: шаблоны проектирования; системы мониторинга; архитектура; требования к системе мониторинга.

Для цитирования: Пасынкова А. А., Викентьева О. Л. Проектирование архитектуры системы мониторинга на основе паттернов проектирования. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 137–150 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–10

1. Introduction

Monitoring systems have become an essential part of various industries, providing real-time information about the health and performance of critical systems. These systems are complex and require sophisticated architectures to handle the data flow, processing, and storage [1]. However, as the systems grow and evolve, they become increasingly challenging to maintain, and changes can have unforeseen consequences [2]. This is where the use of design patterns can be invaluable.

Design patterns are reusable solutions to commonly occurring problems in software design. By applying design patterns, developers can address specific design issues and improve the quality of the system [3-5]. Design patterns have proven to be effective tools in software development, providing solutions to common problems and ensuring that software systems are scalable, maintainable, and flexible [6-8].

The problem is that without using design patterns, the maintenance of monitoring systems can be difficult, time-consuming and prone to errors [9-11]. As the system grows, the complexity increases, and it becomes harder to make changes without causing unintended consequences. Therefore, it is essential to assess the possibility of using design patterns in the development of monitoring system architecture.

Also, the relevance of developing own architecture independently, without using ready-made open-source solutions is justified by the fact that some enterprises cannot do this because of high secrecy and the need to ensure security when working with a monitoring system. Therefore, the use of foreign solutions cannot be chosen.

This article will analyze the possibility of using design patterns to develop the architecture of monitoring systems and provide examples of design patterns that are well-suited to monitoring systems. Во избежание ошибок при форматировании текста статьи настоятельно рекомендуется использовать данный документ в качестве шаблона. Это позволит получать все заданные параметры форматирования текста автоматически. В противном случае необходимо самостоятельно обеспечить выполнение всех требований данного документа (размер страницы, поля и отступы, шрифт, расстояние между колонками и т. д.).

2. Motivation

The motivation for exploring the topic of the use of design patterns in the development of the architecture of monitoring systems comes from the increasing demand for robust and scalable monitoring systems in various industries such as finance, healthcare, and telecommunications. The rapid growth of technology has led to the development of more complex and distributed systems, which require advanced monitoring capabilities to ensure their proper functioning.

However, building a monitoring system that is both scalable and maintainable can be a challenging task. It is difficult to predict all possible scenarios and requirements that the system may face in the future, making it hard to maintain and update the system over time. This is where design patterns come into play. By using proven design patterns, developers can build monitoring systems that are easier to maintain, more flexible, and more scalable [12].

The main goal of this article is to assess the possibility of using design patterns in the development of the architecture of monitoring systems, and to demonstrate their relevance and effectiveness [13-15]. By exploring different design patterns and their applications in monitoring systems, this article aims to provide a comprehensive overview of the benefits of using design patterns in monitoring systems development [16].

This article will be valuable to developers and architects who are involved in the development of monitoring systems, as well as to anyone interested in learning about the benefits of using design patterns in software development.

3. Problem statement

Requirements analysis is an important part of the software development process. It involves collecting and documenting the needs and constraints of stakeholders to ensure that the final product meets their expectations. At this stage, it is necessary to analyze and document the requirements for the monitoring system.

System requirements are the most detailed technical requirements, and they describe how the system will be designed and implemented. System requirements are often expressed in the form of functional and non-functional requirements, and they represent a plan that the development team should follow. System requirements are usually collected during design sessions, technical reviews, and other development processes.

3.1 Functional requirements

Functional requirements describe what the system should do and how it should behave. Examples of functional requirements may include:

1) Data collection and storage: The system should be able to collect data from various sources, such as sensors, devices, and databases, and store them in a centralized location.
2) Data analysis: The system should be able to analyze the collected data and provide information about controlled processes in real time. This can include data aggregation, filtering, and visualization.
3) Alerts and notifications: The system should be able to notify the relevant stakeholders when certain conditions or thresholds are met, for example, when an anomaly or process inconsistency is detected.
4) Reporting and dashboards: The system should provide customized reports and dashboards that allow users to view key performance indicators (KPIs), track progress towards achieving goals and identify areas for improvement.

3.2 Non-functional requirements

Non-functional requirements describe system qualities such as performance, reliability, and security. Next, examples of non-functional requirements will be analyzed:

1) Scalability: The system should be able to handle a large amount of data and users and be able to zoom in and out as needed. Vertical scaling is characterized by an increase in the bandwidth of an individual server or resource, for example, by increasing computing power or memory, which allows you to handle a large load. Horizontal scaling involves adding more servers or resources to handle the increasing load by distributing the workload across multiple machines.
2) Flexibility: The system should be designed in such a way that it can easily adapt to changing requirements without requiring significant changes in its underlying architecture. In the context of monitoring systems, flexibility is important because monitoring requirements can change over time. For example, it may be necessary to add new sensors or devices, as well as to reconfigure the system considering changes in the controlled environment. Flexibility allows for greater maintainability and extensibility.
3) Reliability: The system should be able to work 24/7 without any downtime and provide accurate and reliable data. In the context of monitoring systems, this is important, since any failure can lead to large financial losses, downtime and potentially dangerous situations. One of the ways to achieve reliability is redundancy. Redundancy involves the duplication of critical components or subsystems in the system to ensure that if one component fails, another can take its place. For

example, backup power supplies, network interfaces or data storage devices can be added to the monitoring system to increase reliability. Another way to achieve reliability is fault tolerance, which involves designing the system in such a way that it continues to function even when a component fails. Fault tolerance can be achieved by adding mechanisms such as error detection and correction or automatic failover. In general, reliability engineering involves considering all potential points of failure in the system and developing mechanisms to prevent or mitigate the consequences of these failures.

4) Compatibility: The system must be able to interact with other systems and devices using open standards and protocols. In the context of monitoring systems, compatibility can be used to achieve integration with other software components, devices, or platforms to perform their functions effectively. For example, a monitoring system in a manufacturing facility may need integration with sensors, programmable logic controllers (PLCs) and other industrial automation systems to collect data and perform analysis. The monitoring system must be designed in such a way as to be compatible with these various systems. In addition, the use of standard communication protocols, such as MQTT, REST, can help to implement compatibility between different systems.
5) Maintainability: Maintainability is the ability of a system to remain in good condition over time, which covers all actions related to maintaining and improving the quality of the system, including bug fixes, code refactoring and system updates. The serviced system is easy to understand, modify and expand, and it is less prone to errors and defects.

4. Implementation

The architecture of the platform for intelligent environmental monitoring "Digital Ecomonitoring" is presented using a component diagram (*Fig. 1*).

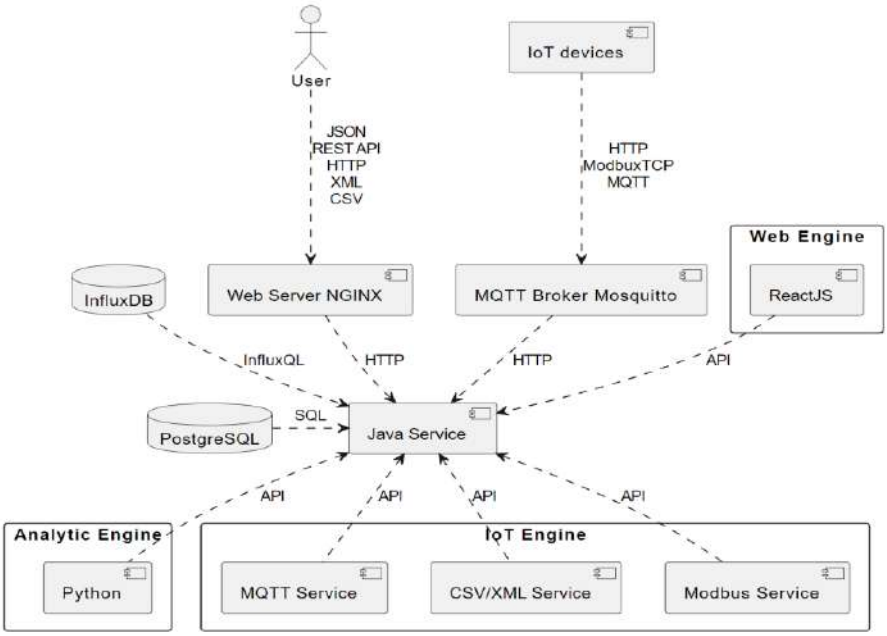


Fig. 1. Component diagram for the platform for intelligent environmental monitoring "Digital Ecomonitoring"

The "Digital Ecomonitoring" platform is designed to provide monitoring and analysis of environmental data in real time, as well as the implementation of emission forecasting. Users also

have the ability to configure alerts based on predefined thresholds, which allows them to take proactive measures in response to environmental changes.

The platform has a multi-level architecture with several components working together. The InfluxDB time series database is used to store measurements read from controllers or uploaded by the user to the platform. The PostgreSQL relational database management system is used to store dashboard and widget settings, accounts and roles, as well as the assignment of access rights. ReactJS is used to create user interfaces in the digital platform. Python is used as an analytical tool for processing data collected by the monitoring system, as well as for predicting values for emissions. NGINX web server is used to process incoming requests from clients and forward them to the corresponding components of the digital platform.

The process of data collection and storage in the Digital Ecomonitoring platform is implemented using the Factory pattern. The abstract Data Collector class is a base class that allows you to create new classes responsible for new sensors without diving into the specific details of their implementation. Data Collector is part of Java Service. In the same way, the abstract Data Storage class is able to create new instances of data warehouses.

The process of data processing, analysis and visualization in the platform is implemented using the Decorator pattern, which allows you to add behavior to a single object without affecting the behavior of other objects in the system. In this case, all additional methods for analysis and forecasting are located in the analytical component implemented by Python [18].

The visualization process in the platform is implemented with an architecture similar to the MVC pattern. In this case, Java Service is a controller that manages communication between databases and ReactJS, which are a Model and a View, respectively [19-20].

The notification process is not clearly expressed in this architecture and is part of the Java Service, which does not allow it to be attributed to any pattern.

For those who want to build monitoring system architecture, there is such a solution as ThingsBoard. ThingsBoard is an open-source solution for IoT platforms. ThingsBoard is used to manage devices, data collection, processing and visualization of collected information. ThingsBoard allows to conveniently organize the process of collecting data from various devices, use a large number of widgets to build informative dashboards that can help with managerial decision-making.

Component diagram for monolithic architecture of ThingsBoard (*Fig. 2*).

The monolithic architecture of ThingsBoard is very popular as it makes it cheaper and faster to develop a monitoring system, which can help to implement it faster. With the help of various protocols, such as HTTP(S), MQTT, CoAP, data enters systems from various devices. Each transport protocol allows to send data to the Rule Engine, which allows devices to change behavior according to the information received, and through the ThingsBoard Core service there is an opportunity to access databases to evaluate the correctness of the information and make appropriate changes. It is assumed that the data collection process is implemented using the Decorator or Factory patterns.

Rule Engine is responsible for processing incoming information according to user-defined logic. It is possible to create a filter, configure alerts when threshold values are reached. This component is responsible for notifying users, which is implemented using the Observer pattern.

The ThingsBoard Core component is responsible for calling the corresponding APIs, managing via WebSocket and tracking the status of connecting devices to the developed system. This component allows to implement devices, users, management rules and connections in the system. It uses the gRPC framework to interact with other components. Also, interaction with databases for storing the received information is implemented through this component, and represents one of the following patterns by architecture: Factory or Decorator.

The ThingsBoard Core component is responsible for processing, analysis and forecasting, the implementation of which also corresponds to the Decorator or Factory patterns.

External systems can receive information from the system using the Rule Engine, which uses gRPC to transfer data to external systems, process data and create processing reports for visualization in ThingsBoard.

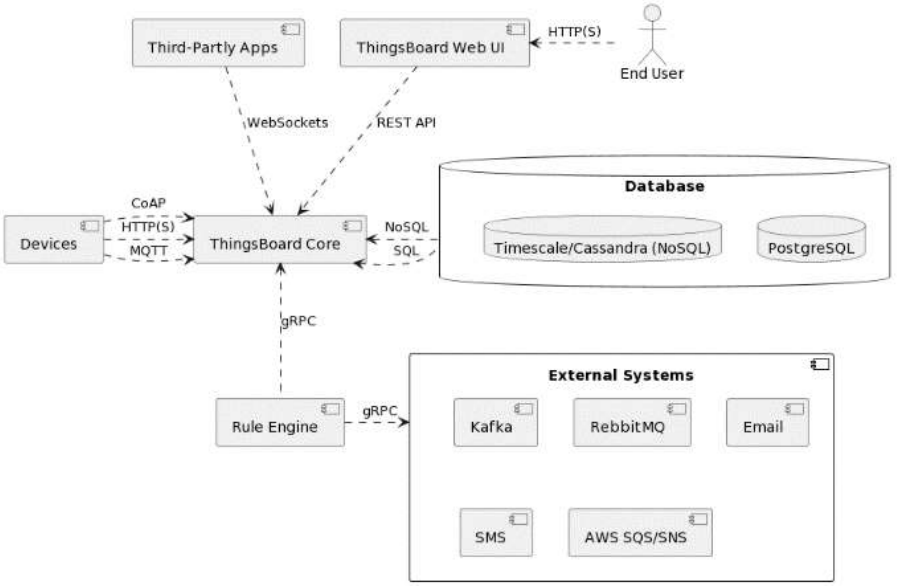


Fig.2. Component diagram for monolithic architecture of ThingsBoard

To organize visualization with the presented system, the MVC pattern is used, which is represented by the following components: Controller – ThingsBoard Core, View – ThingsBoard Web UI, Model – Database.

Component diagram for microservices architecture of ThingsBoard (*Fig. 3*).

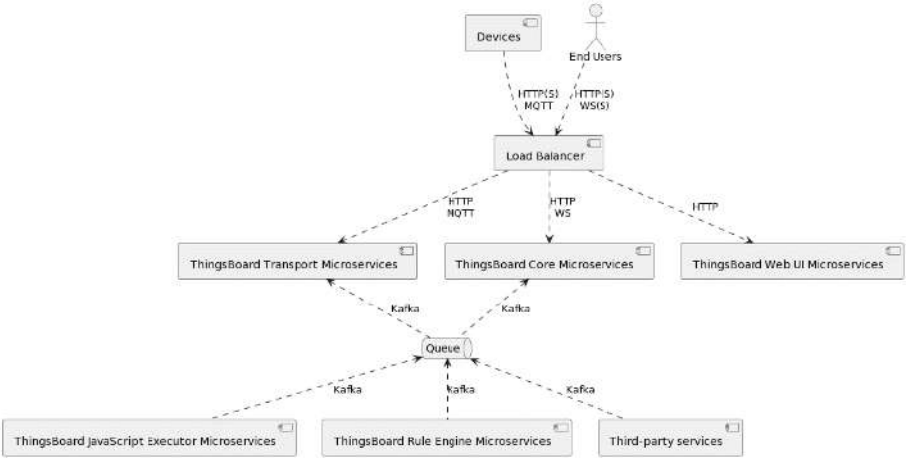


Fig.3. Component diagram for microservices architecture of ThingsBoard

The microservices architecture of ThingsBoard allows to implement a monitoring system with greater flexibility and maintainability. Data from devices is collected using HTTP(S) and MQTT protocols through the corresponding components that are part of Load Balancer. Then the data is

sent to the corresponding services, which transmit them further to other services, process or visualize for users in the system itself.

The applied patterns for the implementation of the monitoring system necessary for the functioning remain the same as for the monolithic architecture, but now there is a separation between the components implementing them into various services, which contributes to easy scalability and increased maintainability.

After analyzing component diagrams for various monitoring systems, a universal component diagram for monitoring systems was designed, which can help in designing your own monitoring system architecture (*Fig. 4*).

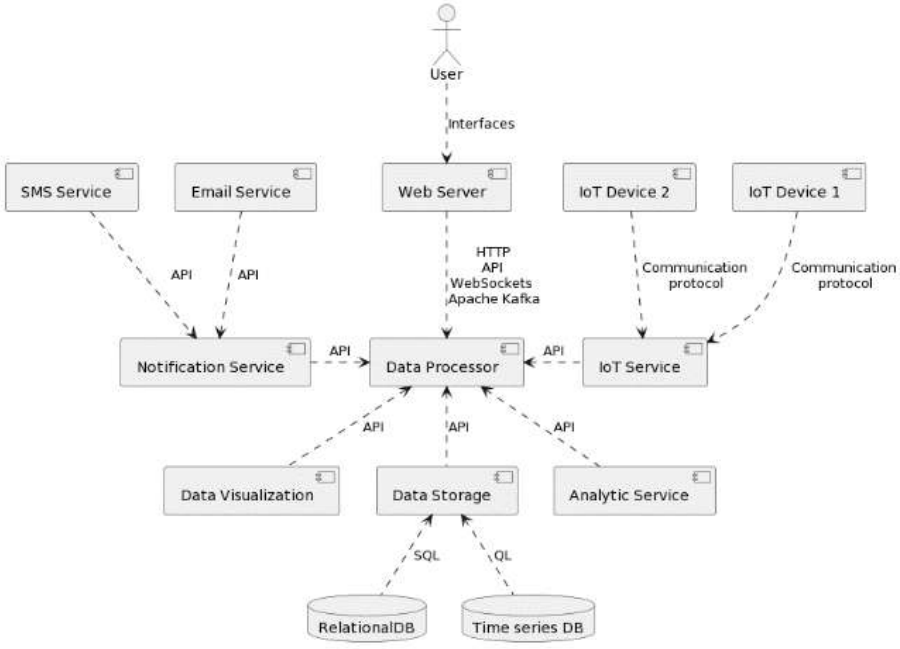


Fig.4. Component diagram of the monitoring system architecture

In the diagram presented, you can see that the system is composed of microservices, which ensures stable operation, maintainability and easy scalability of the monitoring system. The user communicates with the system via a Web Server, so that the Data Processor component knows exactly what the user wants to do.

The list of Data Processor functions also includes communications with Analytic Service, IoT Service, Data Visualization, Data Storage and Notification Service. Analytic Service organizes the analysis and forecasting of the data available in the system. IoT Service communicates with different IoT devices that the monitoring system is connected. Data Visualization displays the data in user-friendly format. Data Storage stores the data in the monitoring system. Notification Service is responsible for informing users of the exceedance of thresholds or for regularly communicating the status of the monitoring system and related objects.

The process of data collection and storage for the monitoring system, implemented using the Factory pattern, it presented using the class diagram (*Fig. 5*).

On the class diagram, there are several abstract classes that allow to easily add new elements to the monitoring system without making changes to its structure. So, Data Collector defines the methods that will be used when implementing specific Collector classes. And Data Storage records what

functional features databases connected to the monitoring system, both relational and time series databases should have.

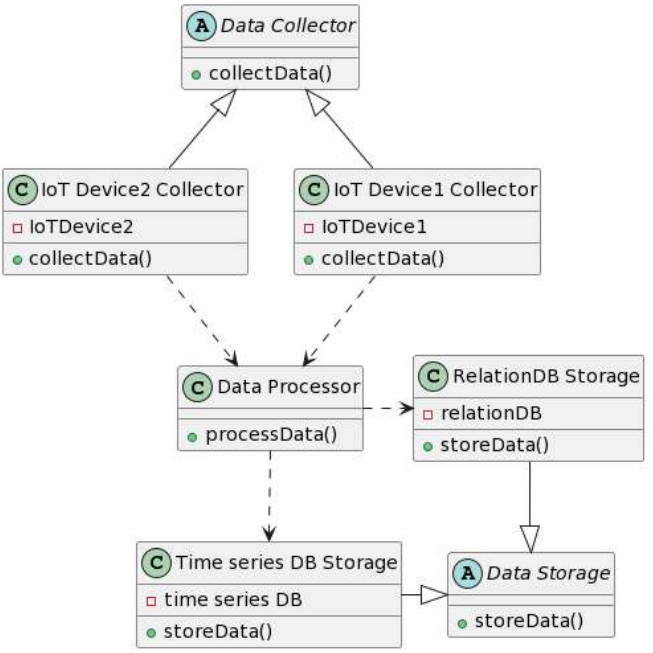


Fig.5. Class diagram of the process of data collection and storage for the monitoring system

Sequence diagram of the process of data collection and storage for the monitoring systems (*Fig. 6*).

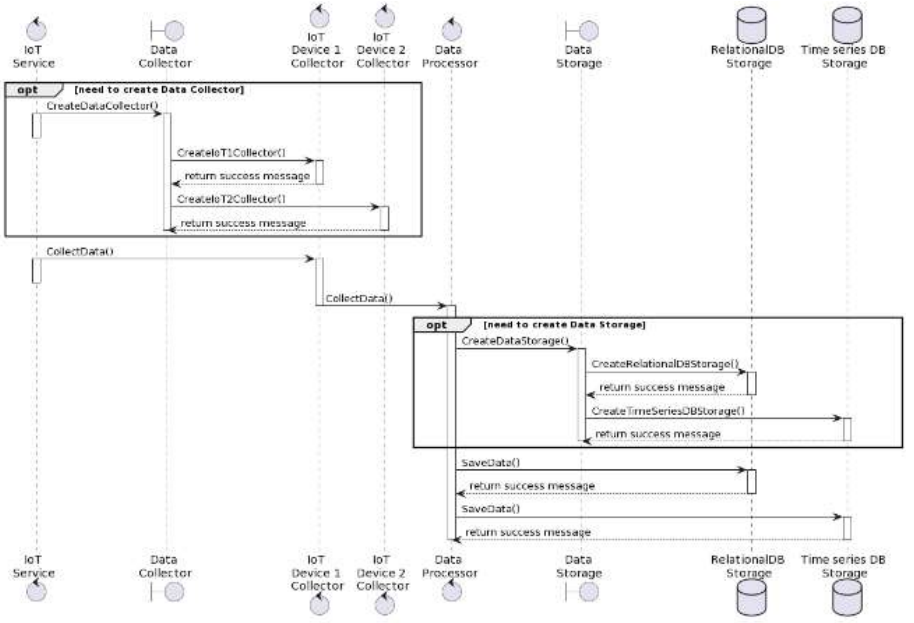


Fig. 6. Sequence diagram of the process of data collection and storage for the monitoring system

In the sequence diagram shown above, there is not only the process of data collection and storage, but also the creation of instances from an abstract base class that implement the appropriate collection method or database to save the collected data.

The process of analyzing and predicting data in the system can be implemented using the Decorator pattern that will allow to add behavior to a separate object without affecting the behavior of other objects in the system. Thus, it's possible to add new methods for data processing and forecasting without the risk of disabling existing methods.

Class diagram of the process of analyzing and forecasting data for the monitoring system (*Fig. 7*).

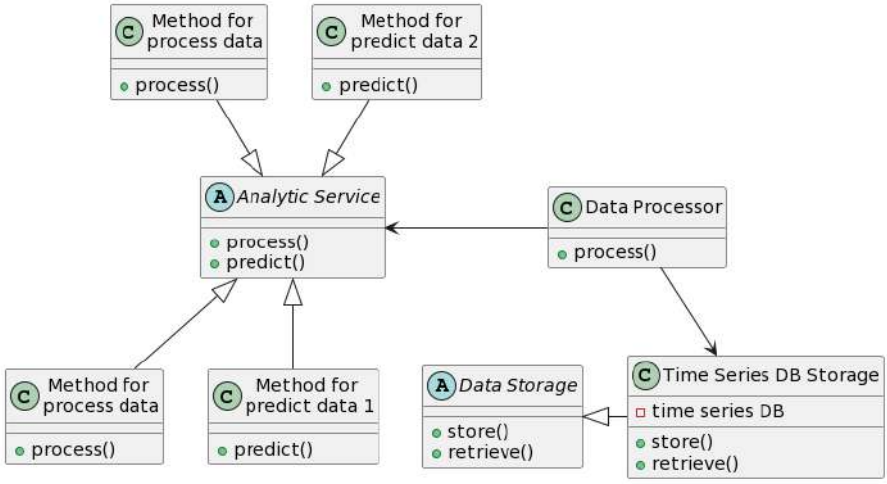


Fig. 7. Class diagram of the process of analyzing and forecasting data for the monitoring system

Methods for data processing and forecasting are extended using the Decorator pattern using the basic abstract class Analytic Service. Similarly, the Time Series DB Storage class is implemented, created according to the abstract Data Storage class.

Sequence diagram of the process of analyzing and forecasting for the monitoring system (*Fig. 8*).

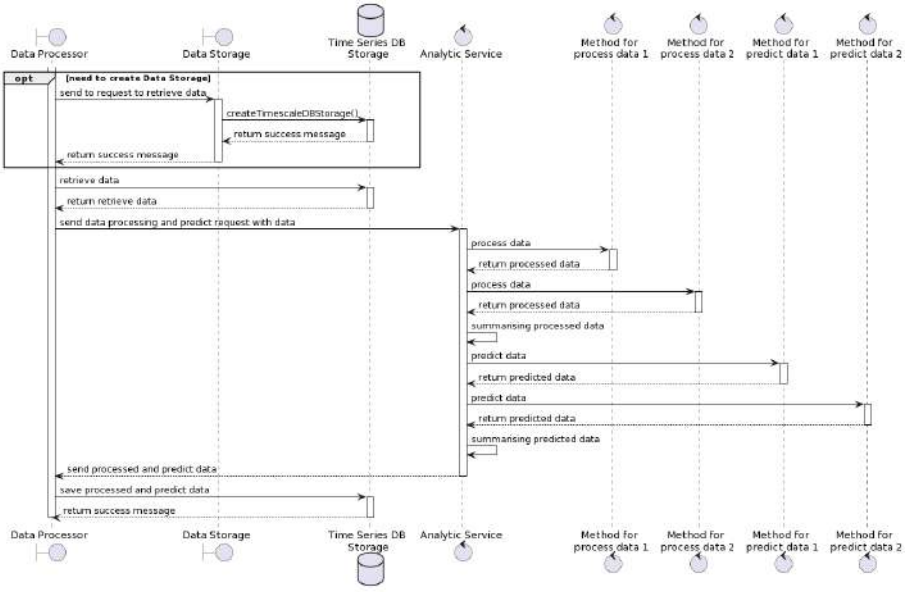


Fig. 8. Sequence diagram of the process of analyzing and forecasting data for the monitoring system

The diagram shows the process of data processing and forecasting, the process of which begins with the creation of a data warehouse according to the abstract base class Data Storage for a time series database. If such a database exists, the Data Processor immediately accesses the database and extracts the necessary information. The received information is sent to the Analytic Service, when it is processed and forecasted using previously established methods in the same way.

The visualization process can be performed using an MVC pattern. This can help to simplify maintenance and system updates. Using this pattern can help achieve separation of the tasks.

Class diagram of the process of visualization data for the monitoring system (*Fig. 9*).

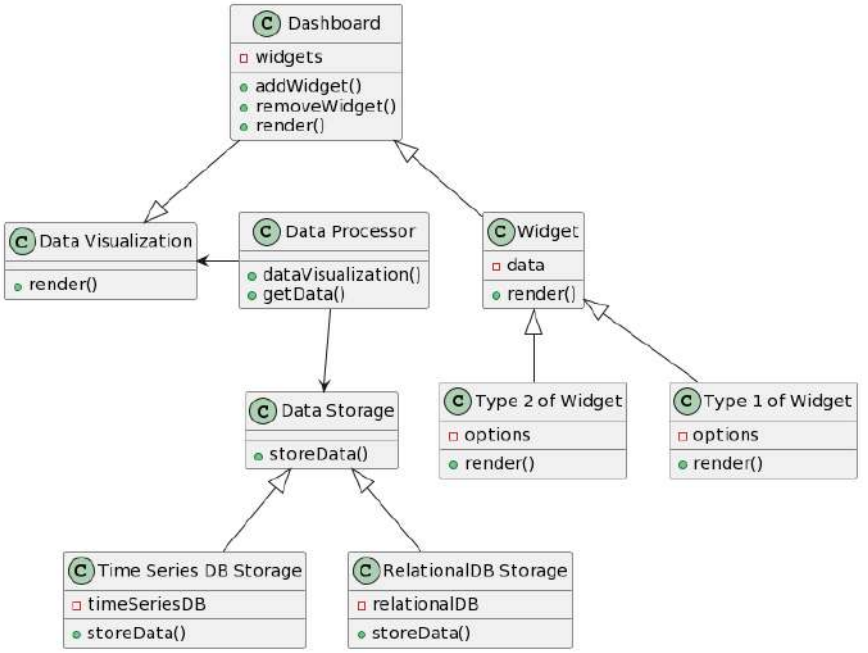


Fig.9. Class diagram of the process of visualization data for the monitoring system

In this case, the Data Processor will be a Controller that will interact between the Model and the View, which are represented by Data Storage and Data Visualization, respectively. The Model is a database repository that can support both relational databases and time series databases, the View is associated with the Dashboard class, which implements widgets defined in the dashboard system.

Sequence diagram of the process of visualization data for the monitoring system (*Fig. 10*).

The diagram shows the interaction of the elements of the system built according to the MVC pattern.

The process of notifying users in the monitoring system can be implemented using the Observer pattern. This pattern allows you to update the values of related objects when the observed objects change [17-18].

Class diagram of the process of notification users for the monitoring system (*Fig. 11*).

This diagram shows the process of notifying users by applying the Observer pattern, which allows to support instantons change in the state of an object with changes in the observed objects.

Sequence diagram of the process of notification users for the monitoring system (*Fig. 12*).

In this sequence diagram, the process of notifying users of the monitoring system occurs when the values received from IoT devices exceed the set range of acceptable values. Data Processor, Data Storage and the databases themselves change their state when updates are required from IoT devices. Also, the Notification Service can change its state in those situations when it is necessary to notify the system user of the events that are taking place.

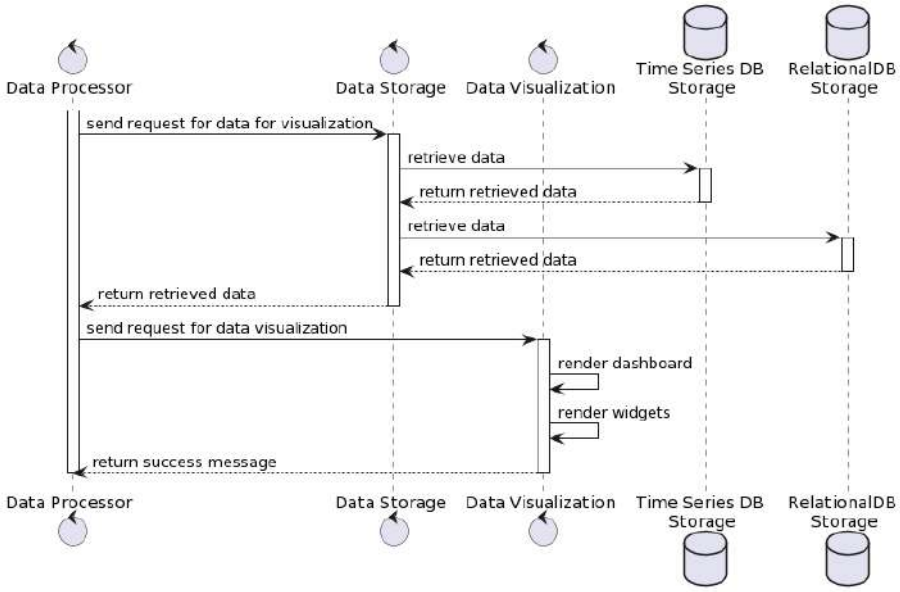


Fig. 10. Sequence diagram of the process of visualization data for the monitoring system

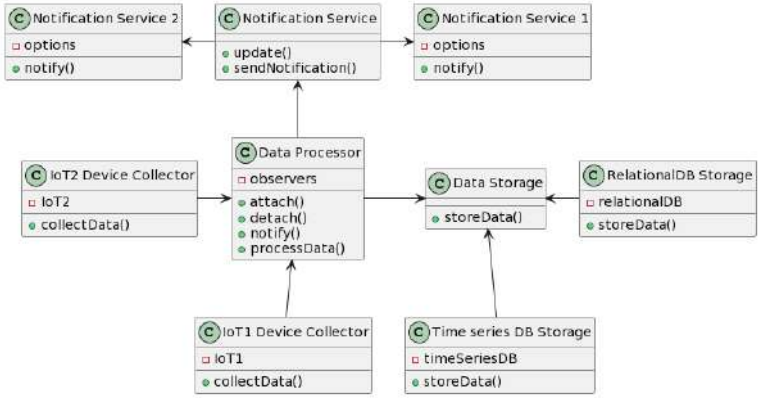


Fig. 11. Class diagram of the process of notification users for the monitoring system

5. Evaluation

The design of the monitoring system architecture depends on the non-functional requirements that will need to be implemented. The following is a list of patterns that can implement the non-functional requirements listed above.

1) Observer pattern: to implement reliability and maintainability by monitoring the state of the object and notifying its dependent elements of any changes.
2) Decorator pattern: to implement vertical scaling, flexibility and maintainability in order to dynamically add functionality to an object without affecting the behavior of other objects.
3) Factory pattern: to implement vertical scaling, flexibility and maintainability in order to create objects without specifying the exact class of the object to be created
4) Microservices pattern: to implement horizontal scaling.
5) Model-View-Control (MVC): to achieve maintainability dividing into three main components: the model, view and control.

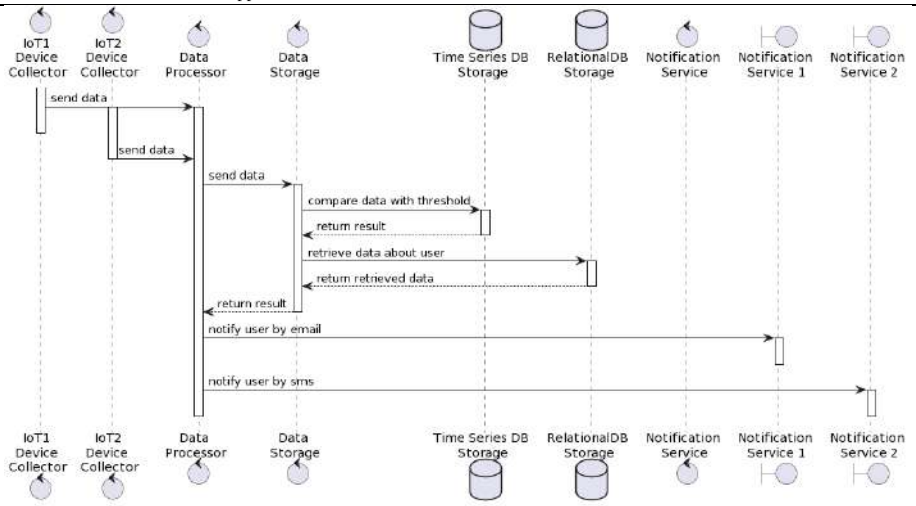


Fig. 12. Sequence diagram of the process of notification users for the monitoring system

6. Conclusion

1) *Data collection and storage*. The monitoring system should collect data from various sensors and devices, process them and store them in a database for further analysis. The Decorator or Factory patterns can be used to create objects representing different types of data.
2) *Data analysis and processing*. Once the data is collected, the monitoring system needs to analyse it to extract meaningful information. The Decorator or Factory patterns can be used to add new analysis capabilities to the system without changing the existing structure.
3) *Data visualization*. The monitoring system should present the data in a clear and understandable form for the user. The Model-View-Controller (MVC) pattern can be used to separate data from the user interface, allowing developers to create different representations of the same data without affecting the underlying data model.
4) *Notifying users about problems*. The monitoring system should notify users when certain conditions are met, for example, when the sensor detects an abnormal value or when the device goes offline. The Observer pattern can be used to trigger alerts when certain events occur.

Conclusion: By considering and implementing best practices and design patterns, it is possible to ensure that the architecture of the monitoring system is scalable, flexible and easy to maintain. This will allow the system to effectively meet the needs of the organization over time as monitoring requirements change.

References

[1]. D. Gurdur et al., 'Knowledge Representation of Cyber-physical Systems for Monitoring Purpose', Procedia CIRP, 2018, vol. 72, pp. 468–473.
[2]. Соснин П.И. Архитектурное моделирование автоматизированных систем: учебник / П.И. Соснин. – Санкт-Петербург: Лань, 2020. – 180 с.
[3]. N. Nazar, A. Aleti, and Y. Zheng, 'Feature-based software design pattern detection', Journal of Systems and Software, 2022, vol. 185, pp. 1–12.
[4]. D. Yu, P. Zhang, J. Yang, Z. Chen, C. Liu, and J. Chen, 'Efficiently detecting structural design pattern instances based on ordered sequences', Journal of Systems and Software, 2018, vol. 142, pp. 35–56.
[5]. S. K. Lo, Q. Lu, L. Zhu, H.-Y. Paik, X. Xu, and C. Wang, 'Architectural patterns for the design of federated learning systems', Journal of Systems and Software, 2022, vol. 191, p. 111357.
[6]. J. Arm, Z. Bradac, O. Bastan, J. Streit, and S. Misik, 'Design pattern for the runtime model-based checking of a real-time embedded system', IFAC-PapersOnLine, 2019, vol. 52, no. 27, pp. 127–132.

[7]. Z. Moudam and N. Chenfour, 'Design Pattern Support System: Help Making Decision in the Choice of Appropriate Pattern', Procedia Technology, 2012, vol. 4, pp. 355–359.
[8]. F. Pfister, V. Chapurlat, M. Huchard, and C. Nebut, 'A Design Pattern meta model for Systems Engineering', IFAC Proceedings Volumes, 2011, vol. 44, no. 1, pp. 11967–11972.
[9]. A. Ampatzoglou, O. Michou, and I. Stamelos, 'Building and mining a repository of design pattern instances: Practical and research benefits', Entertainment Computing, 2013, vol. 4, no. 2, pp. 131–142.
[10]. J. Dong, D. S. Lad, and Y. Zhao, 'DP-Miner: Design Pattern Discovery Using Matrix', in 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07), Tucson, AZ, USA: IEEE, Mar. 2007, pp. 371–380.
[11]. A. Ampatzoglou, G. Frantzeskou, and I. Stamelos, 'A methodology to assess the impact of design patterns on software quality', Information and Software Technology, 2012, vol. 54, no. 4, pp. 331–346.
[12]. Шаблоны проектирования программного обеспечения киберфизических систем зданий / А.В. Кычкин [и др.] // Прикладная информатика. – 2020. – Т. 15. – № 86. – С. 48-62.
[13]. C. Liu and P. Jiang, 'A Cyber-physical System Architecture in Shop Floor for Intelligent Manufacturing', Procedia CIRP, 2016, vol. 56, pp. 372–377.
[14]. J. E. Correa, R. Toro, and P. M. Ferreira, 'A new paradigm for organizing networks of computer numerical control manufacturing resources in cloud manufacturing', Procedia Manufacturing, 2018, vol. 26, pp. 1318–1329.
[15]. S. J. Oks, M. Jalowski, A. Fritzsche, and K. M. Moslein, 'Cyber-physical modeling and simulation: A reference architecture for designing demonstrators for industrial cyber-physical systems', Procedia CIRP, 2019, vol. 84, pp. 257–264.
[16]. M. M. Hamdan, M. S. Mahmoud, and U. A. Baroudi, 'Event-triggering control scheme for discrete time Cyberphysical Systems in the presence of simultaneous hybrid stochastic attacks', ISA Transactions, 2021, vol. 122, pp. 1–12.
[17]. J. Hu, W. Wu, F. Zhang, T. Chen, and C. Wang, 'Observer-based dynamical pattern recognition via deterministic learning', Neural Networks, 2023, vol. 159, pp. 161–174.
[18]. K. Aljasser, 'Implementing design patterns as parametric aspects using ParaAJ: The case of the singleton, observer, and decorator design patterns', Computer Languages, Systems & Structures, 2016, vol. 45, pp. 1–15.
[19]. B. V. Ivanovich, B. V. Vladimirovich, N. F. Victorovich, B. V. Viktorovich, and A. L. Vitalievna, 'Using MVC pattern in the software development to simulate production of high cylindrical steel ingots', Journal of Crystal Growth, 2019, vol. 526, p. 125240.
[20]. A. Sunardi and Suharjito, 'MVC Architecture: A Comparative Study Between Laravel Framework and Slim Framework in Freelancer Project Monitoring System Web Based', Procedia Computer Science, 2019, vol. 157, pp. 134–141.

Информация об авторах / Information about authors

Александра Алексеевна ПАСЫНКОВА – магистр НИУ ВШЭ на специальности «Информационная аналитика в управлении предприятием».

Alexandra Alekseevna PASINKOVA holds a Master's degree from the Higher School of Economics in the specialty "Information Analytics in Enterprise Management".

Ольга Леонидовна ВИКЕНТЬЕВА – доцент кафедры информационных технологий в бизнесе на факультете социально-экономических и компьютерных наук в НИУ ВШЭ в Перми. В сферу научных интересов входят: CASE-технология, Анализ и моделирование бизнес-процессов, объектно-ориентированное программирование, объектно-ориентированное моделирование, проектирование систем, управление проектами, активные методы обучения.

Olga Leonidovna VIKENTYEVA is a docent of the Department of Information Technology in Business at the Faculty of Socio–Economic and Computer Sciences at the HSE in Perm. Her research interests include: CASE technology, Analysis and modeling of business processes, object-oriented programming, object-oriented modeling, system design, project management, active learning methods.

DOI: 10.15514/ISPRAS-2023-35(3)-11

Finding More Bugs with Software Model Checking using Delta Debugging

[1,2] *O.M. Petrov*, ORCID: 0009-0004-6245-9615 <*o.petrov@ispras.ru*>

[1] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[2] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Many verification tasks in model checking (one of the formal software verification approaches) can't be solved within bounded time requirements due to combinatorial state space explosion. In order to find a bug in the verified program in a given time, a simplified version of it can be analyzed. This paper presents DD** algorithms (based on the Delta Debugging approach) to iterate over simplified versions of the given program. These algorithms were implemented in software-verification tool CPAchecker. Our experiments showed that this technique might be used to find new bugs in real software.

Keywords: formal software verification; software model checking; delta debugging; CPAchecker.

For citation: Petrov O.M. Finding More Bugs with Software Model Checking using Delta Debugging. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 3, 2023. pp. 151-162. DOI: 10.15514/ISPRAS-2023-35(3)-11

Acknowledgements. The author thanks his colleagues Anton Vasilyev and Vadim Mutilin for their useful advices on the article topic.

Поиск новых ошибок методом верификации моделей с помощью подхода дельта-отладки

[1,2] *О.М. Петров,* ORCID: 0009-0004-6245-9615 <*o.petrov@ispras.ru*>

[1] *Московский государственный университет имени М.В. Ломоносова,*
Россия, 119991, Москва, Ленинские горы, д. 1.
[2] *Институт системного программирования им. В.П. Иванникова РАН,*
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Зачастую инструмент формальной верификации моделей программ не может получить вердикт за ограниченное время из-за комбинаторного взрыва пространства состояний. Чтобы найти ошибки в верифицируемой программе за выделенное время, может быть проанализирована упрощённая её версия. В этой работе представлены алгоритмы DD**, основанные на подходе Delta Debugging, с помощью которых производится перебор упрощённых версий программы. Эти алгоритмы были реализованы в инструменте статической верификации программ CPAchecker. Наши эксперименты показали, что предложенный метод может быть использован для нахождения ошибок в программных системах, используемых на практике.

Ключевые слова: формальная верификация программ; верификация моделей; delta debugging; CPAchecker.

Для цитирования: Петров О.М. Поиск новых ошибок методом верификации моделей с помощью подхода дельта отладки. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 151–162 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–11.

Благодарности. Автор благодарит своих коллег А.А. Васильева и В.С. Мутилина за советы по теме статьи.

1. Introduction

A significant portion of tasks and problems today are solved with the aid of software. With the increase in the scale and complexity of tasks, the scale and complexity of the software systems that solve them increase, as does the difficulty of preventing, detecting, and eliminating errors in them.

Approaches to detecting errors in programs can be divided into three types: expertise, dynamic analysis, and static analysis. Expertise is the manual review of code (or other development artifacts) by a human with a high enough level of expertise and is not scalable. Dynamic analysis methods involve the analysis of a sufficiently long run of the software system or the analysis of test runs. It can be automated, but it can only detect bugs on paths that were included in the test suite and cannot prove program correctness.

Static analysis includes methods for analyzing the source or binary code of a program without running the program. Lightweight static analysis techniques such as control flow analysis and data flow analysis are thoroughly used in compilers [1] and can be used to detect probable defects in a short time. On the other hand, formal verification methods make it possible to reliably obtain evidence of an error (counterexample) or even prove the absence of errors (correctness of a program with respect to a given formal specification), but this may require significant computational resources or human aid. One of the most successful tools for automatic model checking of C programs is CPAchecker[1] [2], [3]. With its help, several hundred errors were found in the code of the Linux operating system drivers[2] [4], [5].

The tool is actively developed and wins medals in the software verification competitions SV-COMP several years in a row [6]–[8].

Although at the SV-COMP 2022 competition this tool received second place in the summary category Overall, it was unable to complete the verification of a considerable number of programs due to a 15-minute CPU time limit. Table 1 compares the CPAchecker verification tool and the winners in the corresponding competition categories in terms of the number of programs that were verified within the allotted time.

The table shows that even the winners in the respective categories failed to verify a significant portion of programs, especially in the SoftwareSystems category, which consists of complex programs that are close to the real software systems used. The obvious solution to the lack of resources for verification is to allocate more resources, but often this does not help to get a verdict.

In this work, we use the approach of simplifying the verified program. This approach is known, but we have proposed an automatic approach to the systematic enumeration of simplified versions of the program. For this, algorithms based on the Delta Debugging algorithm are proposed. The implementation manipulates (removes) function bodies from the internal representation of the program in CPAchecker, a control flow automation.

The proposed enumeration of simplified program versions takes a significant amount of time, and the technique's limitations lead to the loss of up to 38%[3] of verdicts that the baseline analysis could find. However, this way it is possible to get an *unsafe* verdict for the 32% of the programs, for which respective baseline analysis can not obtain a verdict in the same amount of time. Due to the complexity of proving the correctness of the original program on the basis of the correctness of simplified programs, the search for *safe* verdicts remains outside the scope of this work.

[1] https://gitlab.ispras.ru/verification/cpachecker
[2] http://linuxtesting.org/ldv
[3] See evaluation on Linux USB drivers in section 4.2.

2. Related work

The following two subsections describe techniques that can be applied in model checking in order to obtain results: specific to the problem of combinatorial explosion in model checking, general-purpose techniques for reduction of the software to be verified, and reuse of partial results of verification. The third subsection describes Delta Debugging approach that is used to enumerate simplified versions of the program to be verified.

Table 1. Programs verified, SV-COMP 2022.

Category	Programs in category	Verified by CPAchecker	Winner in category	Verified by winner
ReachSafety	5400	3477 (64%)	VeriAbs	4476 (83%)
MemSafety	3321	2992 (90%)	Symbiotic	3264 (98%)
ConcurrencySafety	763	377 (49%)	Deagle	559 (74%)
NoOverflows	454	369 (81%)	CPAchecker	—//—
Termination	2293	1023 (45%)	UAutomizer	1589 (69%)
SoftwareSystems	3417	1830 (54%)	Symbiotic	1261 (37%)
FalsificationOverall[a]	13355	3726 (28%)	CPAchecker	—//—
Overall[b]	15648	10195 (65%)	Symbiotic	8962 (57%)

[a]All previous categories except Termination.
[b]All previous categories including Termination.

2.1 Model checking techniques

Model checking is a formal software verification technique, i.e. a program is checked against specification – some formally expressed property (often in a from of a temporal logic formula [9]). Model checker explores state space of the given program and checks seen states against the given specification. The program state represents values of all program variables and the current control location (the value of the instruction pointer).

When a state violates the given specification, model checker can export a *counterexample* – a trace to this state – as a specification violation witness. This ability of systematic search for error paths makes model checkers useful tools for bug-finding.

One of the well-known techniques to reduce generic software model is abstraction. Explicit model of a program is overapproximated by an abstract model in a way that does not lose counterexamples. Abstraction is often paired with counterexample-guided abstraction refinement [10]. This way, model checker starts with the most abstract model; when a *spurious* counterexample is present in the abstract model, but is not feasible in the verified software, it is used to make the abstraction more precise. The abstract model is refined this way until a feasible counterexample is found or the whole model is checked.

Other classic techniques include partial order reduction (taking into account that some asynchronous events simulated in a different order lead to the same state [11]), and symmetry reduction (using symmetry in systems with multiple identical components [12]), both of which are used for model checking of concurrent systems; and symbolic model checking, i.e. using binary decision diagrams as compact encoding of state space [13].

Another well-known technique is bounded model checking [14]. In order to avoid state-space explosion, the length of explored traces in the model is bounded, and therefore model checker either provides a counterexample that is shorter than the imposed limit, or proves that there are no such counterexamples. This technique is thoroughly improved and is used in practice for bug-finding.

2.2 Partial verification and verification of parts

Another way for state space reduction is to reduce the input program that needs to be modelled. This can be done using component-based approach or reusing previous verification results.

Usually large-scale software systems are divided into components. Software verification can benefit off this structure via interface rule, assume-guaranty reasoning, or other techniques oriented on component-based software verification [15]. Contrarily, decomposition of specification can also be useful [16]. Incremental verification [15] and extreme model checking [17] can be used with incremental software system development and extreme programming, respectively. This way software verification benefits from the fact that most part of the software system was already verified, therefore verification of the new version of the software is approachable.

Another technique that is especially useful for regression verification is precision reuse [18]. In similar fashion, the *precision* of abstract model of the software older version can be used to achieve efficient verification of the newer version. Conditional model checking [19] proposes to export partial results of a verification run as a predicate describing safe (explored) part of the verified software and add such predicate as an input to a verification tool. *Safe* verdict is represented as *true*, and *unsafe* verdict is represented as *false*. This way different tools can exchange information.

The state-of-the-art verification tools make it possible in practice to increase the efficiency of verification by transferring information between two tools (or a tool running in different configurations). A tool and language "for the composition of cooperative approaches" have been proposed [20]. At the SV-COMP 2022 competition [7], such a tool could have taken second place in the ReachSafety, MemSafety, and Termination categories and first place in the NoOverflow category, but it did not participate in the rating because it used other participating instruments.

Another well-known approach that can be viewed as program simplification technique is program slicing [21]: only statements that affect values of the given variables at the given instructions through control or data flow remain in program. This technique was evaluated with CPAchecker [22], [23] with mixed results, and was implemented [24] as a *configurable program analysis* inside CPAchecker (i.e. it can be used alongside other CPA to construct and refine an abstract model of a given program [3]).

2.3 Delta Debugging

This paper proposes the automatic enumeration of simplified versions of the program being verified. This technique is closer to the verification of parts of the program. The most known approach to changing input data, program version, or other startup conditions is Delta Debugging, proposed by [25]. These algorithms iterate over subsets of a set of arbitrary homogenous atomic elements that make up the "changeable circumstances". The initial set is split into smaller parts, *deltas*, and for both deltas and their complements the interesting property can be checked. Then deltas are split into ever smaller parts, until they consist of one element.

In this paper, function bodies of an original analyzed program are considered elements, i.e., simplified versions of the same program miss some function bodies. Lines of code, blocks, and operators can also be considered as less coarse elements.

Delta Debugging distinguishes three outcomes in terms of a test run outcome. Let original full set of input elements holds some property *fail* (i.e., test run produces a failure; here, a model checker cannot verify a given program in a given time). Let empty set of input elements (baseline) holds some property *pass* (i.e., test run succeeds; here, a model checker provides a *safe* or *unsafe* verdict, which is the case for an "empty" C program of `int main(){ return 0; }`). These two properties must be mutually exclusive (test cannot succeed and fail simultaneously). The case when neither is held is considered *unresolved* (here, an error occurred in the verification tool). Seminal work proposes three DD algorithms based on the same approach:

- *ddmin*: minimization of fail-inducing subset;
- *ddmax*: maximization of passing subset;
- *dd*: isolation of a fail-inducing difference ("cause").

As these algorithms do not enumerate all of the subsets, the minimum (maximum) found by *ddmin* (*ddmax*) is local. The authors call it 1-minimal (1-maximal), as no element in the found subset can be removed so that *fail* holds (no element can be added so that *pass* holds). When *dd* finds a "cause", that means that there is some "safe" subset for which *pass* holds, but for the "safe" subset together with the "cause" the *fail* holds.

Delta Debugging improvements: The DD algorithms can work with an unstructured set of elements, whether they are commits, user actions, files, lines, HTML tags, tokens, characters. Ignoring the internal structure of the input allows the algorithm to be used in a wide range of situations, but also allows a large number of unnecessary runs due to ignoring information about internal dependencies.

A Hierarchical Delta Debugging (HDD) algorithm has been proposed that is capable of minimizing tree-structured data faster and more effective than *ddmin* [26]. This algorithm uses *ddmin* to minimize each level of the input tree, starting from the root, and removes nodes with their entire subtrees. Authors applied HDD to minimize C programs in form of an abstract syntax tree.

Other improvements and applications of the DD algorithms include subtree hoisting [27] and binary reduction of dependency graphs (e.g. applicable for Java classes) [28].

3. General design

We simplify the verified program (by removing its parts) in order to find an *unsafe* that is also feasible in the original program. Accounting for both of these problems, we need to mutate original program until an *unsafe* occurs; then the resulting counterexample is checked against the restored control flow automaton. If the *unsafe* is confirmed, the algorithm terminates, otherwise the enumeration process continues.

As a result, the following cycle was implemented inside the CPAchecker tool.

1) CPAchecker parses the program and builds its control flow automaton (CFA).
2) CPAchecker starts verification of the program with the time limit specified for one verification round.
3) If a verdict is produced, CPAchecker returns it; otherwise timeout has occured (*fail* outcome in terms of Delta Debugging)[4].
4) If there is no way to mutate the CFA of the program or the time allotted for the whole process has run out, exit with the *unknown* result.
5) Otherwise, change the program CFA. *dd* chooses what to do based on the results of previous verification round.
6) CPAchecker starts verification with the time limit specified for one verification round.
7) If an *unsafe* verdict is produced, check the counterexample.
8) If the counterexample is confirmed against the original program, CPAchecker returns the *unsafe* verdict.
9) Otherwise, go to step 4. For *dd*, *unsafe* and *safe* mean *pass* outcome, and timeout means *fail*.

3.1 Simplification problem

The main question is how to arrange a sufficiently fast enumeration of simplified versions of the program. In the following, we are considering only removing function bodies, as it makes sense to remove coarser elements of the input program before removing more fine-grained elements like blocks and statements, and this case has been implemented and evaluated.

[4] In practice, other problems may occur (such as exceptions thrown by the verification tool), but here we consider only *safe*, *unsafe*, and timeout possible for simplicity.

On the one hand, the more complex the function, the more likely it (or the code that uses it) has a bug. On the other hand, the analysis of complex functions is also resource intensive. In addition, it is worth considering that a large number of simple functions can be worse than a few complex ones.

The complexity of a function can be estimated through the characteristics of its control flow automaton as a graph: the number of vertices, edges, cycles, its cyclomatic complexity, whether there are sink vertices in the function (the possibility of early termination of the entire program); the semantic characteristics of a function as a program: the number of variables, pointers, function calls in it and whether it calls itself, is it a pure function or does it have side effects; finally, how many times the analysis entered certain locations of the function.

The presented problem can be reformulated as the knapsack problem: it is necessary to choose as many interesting (here value is probability of an *unsafe*) functions as possible so that the analysis does not exceed resource constraints (i.e. weight is an estimate of the is complexity of a function for analysis). In such setting, it is enough to enumerate the largest sets of functions, for which the verification completes before the allotted time limit, since smaller subsets of such a set can only miss an *unsafe*. Such a maximum set can be found using Delta Debugging, with timeout being the *fail* outcome, and verdicts *safe* and *unsafe* being the *pass* outcome.

Contrarily, it may be interesting to find a minimum set of functions that can be called a core of complexity, as the verification of this set ends in a timeout. As the *ddmin* algorithm approaches minimum, it tries some of its subsets too, including removing each function from minimum set individually.

Thus, the proposed algorithm for enumerating simplified versions is based on the previously implemented *dd* algorithm, which localizes the cause. Based on it, algorithms *dd*min** and *dd*max** were developed for searching for a suitable configuration by enumeration of minima and, accordingly, maxima.

3.2 Iterative algorithms DD**

The *ddmin* algorithm can be used to find the minimum set of functions each of which is required to reproduce the timeout. Below a *dd*min* algorithm is proposed for finding the minimum set of causes, since we may be interested in the structure of the minimum set of functions, i.e., which functions together form “causes”. *dd*min* showed speed comparable to *ddmin*.

To search for functions without which a timeout does not occur, the *dd* algorithm can be used. The first run of *dd* will split the set of functions into three sets: the set of removed functions, the set of “safe” functions (which the verification tool manages to analyze in the allotted time), and the isolated “cause”, i.e., the set of functions, after adding which to the set of “safe” functions a timeout reappears.

By repeating *dd* on the set of safe functions, we can isolate a new cause among them (and remove some of these functions, adding them to the set of removed functions). *dd* is repeated until the set of safe functions is empty; now we have a set of removed functions and a set of isolated causes, which makes up the minimum program that the verification tool can not verify in the allotted time.

Similarly, you can find the maximum program not with the *ddmax* algorithm, but by iteratively removing causes with *dd*max*. To do this, the cause is deleted after each run, and all the functions that were removed on this run are returned. This way a new cause can be isolated among all other functions. The process continues as long as the timeout continues to occur after the return of the removed functions. Thus, we get a set of causes that have been removed from the program, and a set of safe functions.

It is possible to construct an algorithm that enumerates the optimums based on algorithms that find a local optimum. In the following, two such algorithms, *dd*min** and *dd*max**, are described.

To iterate over minima, it is enough to return all removed functions and remove one of the isolated causes. If the timeout does not occur without this cause, then we return it and try to remove another one. If the timeout reoccurs, then we can find another minimum, since it will not have the cause

that we removed. This way all the causes found can be removed one by one. Similarly, it is enough to add one of the causes to the found maximum to find another maximum by isolating another cause.

Taking into account that *dd*'s complexity with respect to the number of analysis runs performed is linear in the number of considered elements, we obtain, in the worst case, a quadratic dependence on the number of elements. Assuming that the number of causes in the found minimum is bounded from above by some constant, we obtain a linear complexity estimate (with the indicated constant as a factor).

3.3 Counterexample check

CPAchecker has three implementations for checking counterexamples: using CBMC (Bounded Model Checker for C and C++ programs[5]), concrete execution, and using CPAchecker itself. In the first two cases, the found counterexample is exported as a C program. In the latter case, it is exported as a violation witness in the form of a special automaton that directs the analysis along the already found trace [29]. Since translated programs or a violation witness significantly limit the number of possible execution paths of the program, their analysis is much easier than the analysis of the complete original program. Because of that, more complex analyses may be used to confirm *unsafes* found with simple analyses.

When checking a counterexample, it is necessary to correct the representation of the error trace in order to compensate for the fact that it was found on a modified program. For representation as a program, definitions of removed functions have to be added.

To check a counterexample found for a simplified version of the program, the following was implemented. The counterexample is translated into C in much the same way as for CBMC, but the definitions of the removed functions are added to the resulting text. Then re-verification is started from within CPAchecker (by default with the same configuration). Although there is now a potentially complex function, the rest of the program has been simplified to a single trace, so this check requires much less resources compared to the entire program.

4. Evaluation

Two experiments were conducted to evaluate implemented algorithms, both compare *dd*min** and *dd*max** against the baseline CPAchecker analysis with the same CPU time limit. Effectiveness is evaluated as amount of found *unsafes*, efficiency is evaluated as time spent for the tasks.

4.1 A few programs from SV-COMP/ReachSafety

29 programs were chosen arbitrarily for the first experiment from ReachSafety category of the SV-COMP benchmark[6]. These programs are checked for reachability of specified function call (reachable call is considered a bug). 21 of the chosen programs have an error (the call is reachable) and 8 of the programs do not have an error (the call is not reachable). Most of the programs consist of a few functions, some have a lot of branching. For each of the chosen programs, CPAchecker did not provide a verdict in the 2022 competition due to timeout (15 minutes of CPU time).

The time limit was increased from 15 minutes (900 seconds) to 2.5 hours (9000 seconds) of CPU time for verification of one program. The run was performed using BenchExec[7] on a machine with a 16-core 11th generation Intel Core i7-11700 processor at 2.50 GHz, with 32 GB of RAM (of which CPAchecker had allocated 10 MB on the heap and default 1 MB on the stack), and 64-bit operating system Ubuntu 20.04.6 LTS.

[5] http://www.cprover.org/cbmc/

[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

[7] https://github.com/sosy-lab/benchexec

Baseline configuration (`-svcomp22 -benchmark` with extended timelimit) uses sequential combination of different analyses [30]. *dd*min** and *dd*max** configurations used same analyses with time limit of 200 seconds for each verification round.

As seen in *Fig. 1* and *Table 2*, baseline analysis found 6 unsafes (out of 21 programs with an error) and 0 *safes* (out of 8 programs without an error), while both *dd*min** and *dd*max** found only two *unsafes*. For one program with error, an *unsafe* was found by all three configurations. For another program with error, only *dd*min** found an *unsafe*. For yet another one program with error, only *dd*max** found an *unsafe*.

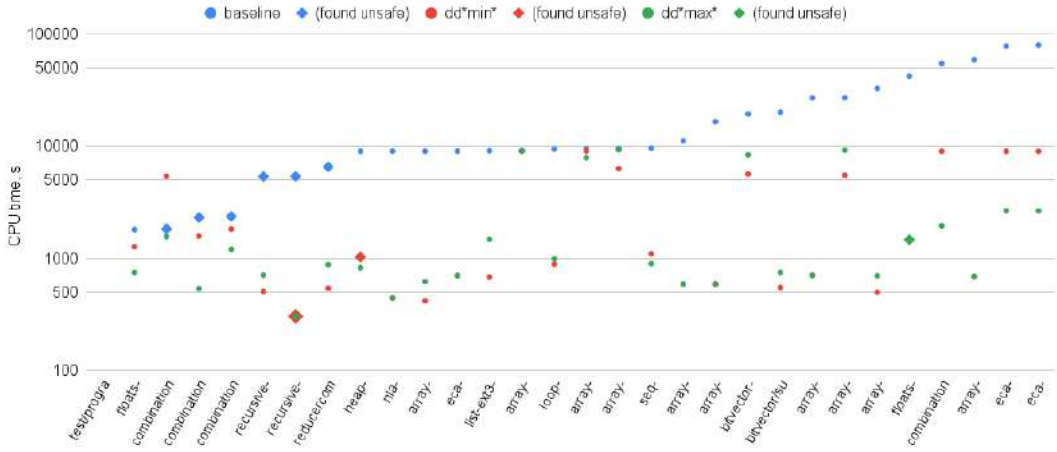


Fig. 1. CPU time for analysis of a few benchmark programs (sorted by baseline time)

Table 2. Results for 29 ReachSafety programs.

	Baseline analysis	***dd*max****	***dd*min****
Total CPU time, h	161	19.0	23.4
Total wall time, h	44.8	7.5	13.3
Safe (8 expected)	0	0	0
Unsafe (21 exp.)	6	2	2
Enumeration completed	—	27	27
Timeout	23	0	0

Small amount of obtained baseline verdicts is not unexpected, as the programs were chosen because CPAchecker could not verify them in time in competition. As these programs consist of small amount of functions, DD** algorithms need more granular elements to manipulate in order to simplify program more precisely and not lose a verdict.

As shown in the table, *dd*min** and *dd*max** in sum took 26% of CPU time of the baseline analysis (46% of wall time).

4.1 Linux USB drivers

In the second experiment, 284 modules of Linux operating system kernel USB device drivers, version 5.10.27, were verified against memory leaks, incorrect dereferences and use after free. It was carried out using Klever system [31] on an 8-core Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS) machine with 32 GB of RAM, and a 64-bit Debian 4.9.246-2 OS.

Baseline analysis configuration (`-smg-ldv`) uses symbolic memory graphs [32].

*dd*min** and *dd*max** configurations used same analysis with time limit of 350 seconds for each verification round.

Fig. 2 shows a quantile graph of the spent CPU time; baseline analysis found 62 *unsafes* (13 of them reqired more than 5 minutes of CPU time), and found 90 *safes* (16 of them required more than 5 minutes of CPU time). Verdict was not produced (result is *unknown*) for other 132 modules:

- for 5 modules, due to encountered recursive functions in module;
- for 100 modules, because of timeout;
- for 6 modules, because more memory was needed;
- for 21 modules, verification was not conducted at all due to a problem outside of verification tool (these are not shown on the figure).

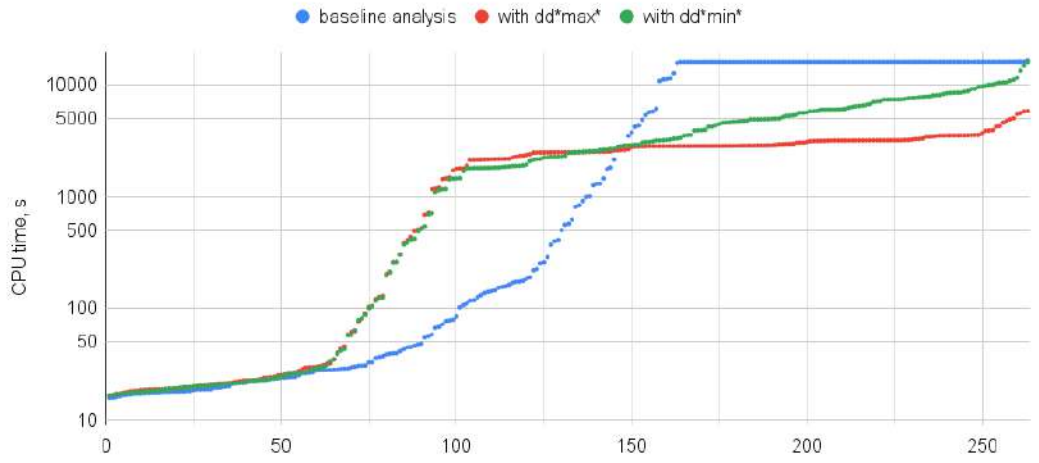


Fig. 2. CPU time for analysis of Linux device driver modules (quantile graph)

It can be seen that for modules whose verification takes 15–35 seconds, the time for the proposed algorithms will most likely also be 15–35 seconds; the time for modules with baseline analysis longer than 35 seconds averages 40–50 minutes for *dd*max** and 40–90 minutes for *dd*min**. Difference under first 350 seconds is explained by the fact that DD** algorithms do not stop verification after first error found, while baseline analysis does. This change in analysis was introduced in order to find all errors that can be present in the original program.

The results for the Linux drivers are presented in *Table 3* and *Table 4*. *dd*max** and *dd*min** obtained 74 and 75 *safe* verdicts, respectively, in cases where verification took less than 350 seconds of CPU time. There was not enough time to verify 100 modules by baseline analysis; there was not enough time for one module to analyze using *dd*min**. For *dd*max** and *dd*min**, the analysis of 130 and 50 modules, respectively, ended because enumeration of simplified versions of the module ended without a verdict.

The *dd*max** algorithm consumed just 29% of the total CPU time (31% of the total wall time) of the baseline. 26 *unsafes* (42% as percentage of *unsafes* obtained by baseline analysis) were found in programs for which baseline analysis can not obtain a verdict.

The *dd*min** algorithm spent 49% of the total CPU time (51% of the total wall time) of the baseline analysis and found 38 *unsafes* (61% as percentage of *unsafes* by baseline analysis) in modules for which baseline analysis can not obtain a verdict.

In total, DD** algoritms obtained new *unsafes* for 42 modules out of 132 modules with *unknown* baseline verdict. Both algorithms obtained an *unsafe* for 23 of these modules.

Change of *safe* to *unsafe* can be explained by incorrect counterexample check: the used analysis does not stop after target state is reached. Additionally, incorrect translation of C enum types induces raise of exceptions.

From the results of the experiments, we can conclude that it may be more effective to use the proposed technique together with a trivial increase of the time limit. For example, simply running the proposed algorithms after the baseline analysis, it is possible to get a linear increase in the number of *unsafes* found (according to the results of the second experiment, 32% of new *unsafes* for additional 29% of total CPU time).

Table 3. Results for 29 Linux USB drivers.

	Baseline analysis	***dd*max****	***dd*min****
Total CPU time, h	493	142	240
Total wall time, h	427	131	218
Safe	90	74	75
Unsafe	62	49	77
Enumeration completed	—	130	50
Timeout	100	0	1
Out of memory	6	3	12
Recursion in module	5	5	5
Other exceptions	0	7	46
Other problems	21	21	21

Table 4. Changed verdicts for Linux USB drivers.

Baseline analysis	***dd*max****			***dd*min****		
	safe	*unsafe*	*unknown*	*safe*	*unsafe*	*unknown*
safe, 90 in total	74	3	13	75	9	6
unsafe, 62 in total	0	20	42	0	30	32
unknown, 132 in total	0	26	106	0	38	94

5. Conclusion

In this paper, the problem of software model checking is considered from the point of view of resource constraints.

Modern methods and approaches for verification of program models were considered. The problem of finding *unsafes* in programs by simplifying the verified program is stated.

Two algorithms, *dd*min** and *dd*max**, were proposed for enumerating simplified versions of programs based on Delta Debugging approach. These algorithms were implemented in the static verification framework CPAchecker, and evaluated on a small set of programs from SV-COMP benchmark and whole set of 5.10 Linux kernel USB device driver modules.

Experiments have shown that the proposed technique takes less than half the total time of baseline analysis and is able to find *unsafes* in programs that are too difficult for baseline analysis, although the total number of verdicts obtained may be less than that of baseline analysis.

There are several directions for a future work: a) program blocks and statements manipulation, b) improvement of counterexample translation, c) reuse of partial results obtained in the analysis of the original program or its simplified versions, d) the optimal time for one round of verification, and e) the optimal order of functions and causes in DD** enumeration.

References

[1]. A. V. Aho, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[2]. D. Beyer and M. E. Keremoglu, “CPAchecker: A tool for configurable software verification,” in Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. Springer, 2011, pp. 184–190.

[3]. D. Beyer, S. Gulwani, and D. A. Schmidt, Combining Model Checking and Data-Flow Analysis. in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, pp. 493–540.

[4]. A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, “Establishing linux driver verification process,” in Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers 7. Springer, 2010, pp. 165–176.

[5]. I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. Novikov, A. K. Petrenko, and A. V. Khoroshilov, “Configurable toolset for static verification of operating systems kernel modules,” Programming and Computer Software, vol. 41, pp. 49–64, 2015.

[6]. D. Beyer, “Software verification: 10th comparative evaluation (SVCOMP 2021),” Tools and Algorithms for the Construction and Analysis of Systems, vol. 12652, pp. 401 – 422, 2021.

[7]. “Progress on software verification: SV-COMP 2022,” in International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2022.

[8]. “Competition on software verification and witness validation: SVCOMP 2023,” in International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2023.

[9]. N. Piterman and A. Pnueli, Temporal Logic and Fair Discrete Systems, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, p. 27–73.

[10]. A. V. Khoroshilov, M. U. Mandrykin, and V. S. Mutilin, “Introduction to CEGAR — counter-example guided abstraction refinement”, Trudy ISP RAN/Proc. ISP RAS, vol. 24, 2013, (in Russian).

[11]. D. A. Peled, Partial-Order Reduction, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018.

[12]. E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla, “Symmetry reductions in model checking,” in International Conference on Computer Aided Verification, 1998.

[13]. S. Chaki and A. Gurfinkel, BDD-Based Symbolic Model Checking, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, p. 219–245.

[14]. A. Biere and D. Kröning, SAT-based model checking, in E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. Handbook of Model Checking, 1st ed. Cham: Springer International Publishing, 2018, ch. 10, pp. 277–303.

[15]. F. Nejati, A. A. A. Ghani, N. K. Yap, and A. B. Jafaar, “Handling state space explosion in component-based software verification: A review,” IEEE Access, vol. 9, pp. 77 526–77 544, 2021.

[16]. S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer, “On-the-fly decomposition of specifications in software model checking,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016.

[17]. T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido, “Extreme model checking,” in Theory and Practice, 2003.

[18]. D. Beyer, S. Lo¨we, E. Novikov, A. Stahlbauer, and P. Wendler, “Precision reuse for efficient regression verification,” in ESEC/FSE 2013, 2013.

[19]. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, “Conditional model checking: a technique to pass information between verifiers,” in SIGSOFT FSE, 2012.

[20]. D. Beyer and S. Kanav, “CoVeriTeam: On-demand composition of cooperative verification systems,” in International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2022.

[21]. M. Weiser, “Program slicing,” IEEE Transactions on Software Engineering, vol. SE-10, no. 4, pp. 352–357, 1984.

[22]. M. Chalupa and J. Strejček, “Evaluation of program slicing in software verification,” in International Conference on Integrated Formal Methods, 2019.

[23]. P. Andrianov, V. Mutilin, M. Mandrykin, and A. Vasilyev, “CPA-BAM-Slicing: Block-abstraction memoization and slicing with region-based dependency analysis,” in Tools and Algorithms for the Construction and Analysis of Systems, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 427–431.

[24]. M. Spiessl, “Configurable software verification based on slicing abstractions,” Master’s thesis, Ludwig-Maximilians-Universita¨t Mu¨nchen (LMU Munich), Mu¨nchen, Germany, Jun. 2018.
[25]. A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” IEEE Trans. Software Eng., vol. 28, pp. 183–200, 2002.
[26]. G. Misherghi and Z. Su, “HDD: hierarchical delta debugging,” Proceedings of the 28th international conference on Software engineering, 2006.
[27]. D. Vince, R. Hodován, D. Bársony, and Á. Kiss, “The effect of hoisting on variants of Hierarchical Delta Debugging,” Journal of Software: Evolution and Process, vol. 34, 2022.
[28]. C. G. Kalhauge and J. Palsberg, “Binary reduction of dependency graphs,” Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019.
[29]. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and T. Lemberger, “Verification witnesses,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 31, pp. 1 – 69, 2022.
[30]. M. Dangl, S. Löwe, and P. Wendler, “CPAchecker with support for recursive programs and floating-point arithmetic,” in Tools and Algorithms for the Construction and Analysis of Systems, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 423– 425.
[31]. E. Novikov and I. Zakharov, “Towards automated static verification of GNU C programs,” in Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers 11. Springer, 2018, pp. 402–416.
[32]. A. A. Vasilyev and V. S. Mutilin, “Predicate extension of symbolic memory graphs for the analysis of memory safety correctness,” Programming and Computer Software, vol. 46, pp. 747 – 754, 2020.

Информация об авторах / Information about authors

Олег Максимович ПЕТРОВ — старший лаборант, магистр факультета вычислительной математики и кибернетики (2023). Его научные интересы включают верификацию моделей программ, delta debugging.

Oleg Maximovich PETROV is a senior laboratory assistant and a master of the Faculty of Computational Mathematics and Cybernetics (2023). His research interests include software model checking, delta debugging.

DOI: 10.15514/ISPRAS-2023-35(3)-12

Framework for Machine Instruction Usage Analysis

D.E. Pechenev, ORCID: 0000-0003-0575-0807 <*d.pechenev@spbu.ru*>
I.A. Kirilenko, ORCID: 0000-0003-4384-8274 <*y.kirilenko@spbu.ru*>
O.A. Afonina, ORCID: 0009-0009-4109-1248 <*o.aphonina@gmail.com*>

St. Petersburg State University,
7-9 Universitetskaya Embankment, St. Petersburg, Russia, 199034

Abstract. When migrating software to new hardware architectures, including the development of optimizing compilers for new platforms, there is a need for statistical analysis of data on the use of different machine instructions or their groups in the machine code of programs. This paper describes a new framework useful for statistical research on machine opcodes that is designed to be extensible and a dataset that can be used by other researchers. We automatically collect data on different GNU/Linux distributions and architectures and provide facilities for its statistical analysis and visualization. Related technical issues are discussed, and solutions to some of them are proposed.

Keywords: RISC-V; software migration; software reengineering; machine code analysis; machine instructions analysis; ISA analysis; opcodes; compiler construction; code optimizations.

For citation: Pechenev D.E., Kirilenko I.A., Afonina O.A. Framework for machine instruction usage analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 163-170. DOI: 10.15514/ISPRAS-2023-35(3)-12

Фреймворк для анализа использования машинных инструкций

Д.Е. Печенев, ORCID: 0000-0003-0575-0807 <*d.pechenev@spbu.ru*>
Я.А. Кириленко, ORCID: 0000-0003-4384-8274 <*y.kirilenko@spbu.ru*>
О.А. Афонина, ORCID: 0009-0009-4109-1248 <*o.aphonina@gmail.com*>

Санкт-Петербургский государственный университет,
199034, Россия, г. Санкт-Петербург, Университетская набережная, д. 7–9.

Аннотация. При миграции программного обеспечения на новые аппаратные архитектуры, включая разработку оптимизирующих компиляторов для новых платформ, возникает необходимость в статистическом анализе данных об использовании различных машинных инструкций или их групп в машинном коде программ. В данной работе описывается новый фреймворк, полезный для статистического анализа машинного кода, который разработан с учетом возможности расширения. Предоставляется набор данных, который может быть использован другими исследователями. Фреймворк позволяет автоматически собирать данные с различных дистрибутивов GNU/Linux и архитектур, а также предоставляет средства для их статистического анализа и визуализации. Обсуждаются связанные с этим технические проблемы и предлагаются решения некоторых из них.

Ключевые слова: RISC-V; миграция ПО; реинжиниринг ПО; анализ машинного кода; анализ машинных инструкций; анализ ISA; опкоды; создание компиляторов; оптимизация кода.

Для цитирования: Печенев Д.Е., Кириленко Я.А., Афонина О.А. Фреймворк для анализа использования машинных инструкций. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 163–170 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–12

1. Introduction

Currently, the open-source RISC-V Instruction Set Architecture (ISA) is actively developing and gaining popularity. According to RISC-V International [1], more than 3,100 RISC-V members across 70 countries contribute and collaborate to define RISC-V open specifications as well as convene and govern related technical, industry, domain, and special interest groups. In September 2022, the RISC-V Alliance was created in Russia [2]. And in December 2022, the European Union released €270 million to build RISC-V hardware and software [3].

In the RISC-V community, the issue of optimizing programs specifically for this architecture is acute. In order to plan software migration, compiler developers and specialists optimizing particular sections of machine code in a non-trivial way manually need to understand which packages and utilities in popular GNU/Linux distributions on various platforms use, for example, vector extensions or instructions for speeding up encryption. This knowledge would help them understand how the compiler can be improved, and in which programs there are sections of machine code that should be optimized manually for the RISC-V architecture.

In this context, it is also necessary to understand how the various GNU/Linux distributions are ready to migrate to RISC-V, that is, how the machine code of their packages is optimized and able to perform tasks in an efficient manner.

To achieve this, a statistical analysis of the machine code is essential, namely, an analysis of the use of different types of machine instructions in the program code. However, the described problems are far from the only cases when such an analysis would be useful. Another example is when the compiler developer needs to find out how the generated machine code of programs has changed in general after changes in the compiler. This technique in particular is planned to be used

to assess the quality of firmware optimization in embedded systems, for example, routers and data warehouses.

In this paper, we describe a new framework that makes it easy to answer such questions. On the one hand, it allows one to automate the collection of data on the machine instruction usage on different GNU/Linux distributions and architectures, and on the other hand, it provides a wide range of tools for statistical analysis and visualization of this data. We also demonstrate how using our framework one can get the main results of [4] and show our advantages over it.

The framework code is open and posted on GitHub [5].

2. Background

As described in [4], "*static analysis* applies to a binary at rest, and, in this case, operates on a disassembled instruction stream. In contrast, *dynamic analysis* observes an executing application using hardware traps and debug instructions, or analyzes an instruction trace gathered during prior execution."

The quality and completeness of dynamic analysis depend on the representativity of the input data provided to the application. Given the fact that even for a single application collecting such data requires a thorough and time-consuming analysis of the code and execution graph, it becomes clear that obtaining such a set of representative inputs for all of the analyzed packages is impossible. This effectively rules out dynamic analysis. Taking that into account, this article considers only static analysis of the machine code.

We need the framework to be able to help answer various questions: discover the most popular and rare instructions, find out exactly where in certain packages specific instructions that accelerate the program are used, compare usage of various types of machine instructions between different GNU/Linux distributions and platforms and a lot more. In this paper, we show that to attain this goal, the instructions and their number are sufficient as data to be collected from a specific file. Nevertheless, the question may arise how to analyze such data when there are so many instructions, and some of them do fundamentally the same thing, for instance, `movl`, `movw`, and `movb`. We also propose an approach that will help to cope with this problem for the x86-64 architecture.

Considering that GNU/Linux is the de facto standard in the community of developers from all over the world, we make the framework work with this operating system. However, we require it to be able to collect data from different distributions, since they are compiled with various options, which makes the machine code of their packages distinct. Moreover, we demand that the framework is capable of working with different ISA, since the same packages can be optimized in various ways for different architectures. This is particularly important for performance critical code optimized manually in such software as archivers, video codecs, machine learning libraries, and so on.

3. Related work

The idea of applying static analysis of machine code is not novel. As an example, it is widely used in malware detection tasks. In article [6], the frequencies of 29 opcodes chosen by the author are used as features to train Random Forest, AdaBoost, XGBoost, and Voting Classifier-based models for detecting malicious executable files. Another research [7] reveals the relationship between the rarest instructions and code maliciousness. However, the data was collected from a few files, which is not enough for complete analysis. Moreover, data of the research is not publicly available.

Instruction frequencies have also been used in works [8]–[10] to determine not only the maliciousness of executable files, but also their belonging to virus families. Firstly, the opcodes with the highest predictive value were identified [8] using 8 evaluation metrics. The authors of the paper found out that it is possible to reduce the number of features (opcodes) from 443 to 180 without loss of accuracy and to 10 with 94.2% accuracy. The analysis was performed for 5 families of crypto-ransomware for Windows.

Secondly, it is shown [9] that histograms of instruction frequencies can help classify a family of metamorphic viruses. A set of such histograms was collected for the NGVCK family of viruses and an average histogram was constructed for it. The frequencies were obtained by counting the operation codes of the instructions in the disassembled binaries (PE, COFF). The classification is based on the calculation of the Minkowski distance for the histograms. The proposed method was tested on only 100 files, and only one family of viruses was considered, so it is not possible to claim its effectiveness.

Thirdly, distribution of instruction usage frequencies is used [10] to quickly classify and detect malware with low computational cost. For ELF binary files, sequences of instructions, sorted by frequency of use, are constructed, and the number of intersections where edges join the same instructions in the resulting sequences is counted. Depending on the range in which the number of crossovers lies, we can assume whether the program is malicious and belongs to the family of viruses in question. The results obtained in the study are pretty encouraging, but the ideas were not tested on a large dataset, and no reliable metrics for the proposed classification were presented.

Besides, instruction n-gram (a contiguous sequence of n items) frequencies are used to determine the maliciousness of executable files. In [11], opcode n-gram patterns are used as features for the classification process. The authors conduct experiments to identify the representation of n-grams, sizes of n-grams, ways to select them for using as features, and the best classifier. The 2-gram opcodes outperformed all others, and DF proved to be the best feature selection method.

Paper [12] explores methods to detect malware based on machine instruction behavior. The authors propose an approach to extract instruction sequences based on the control tree. The decompiled executables are not only analyzed as text files: all kinds of program execution paths are constructed for them, which are concatenated to produce a flow of operations. The frequency statistics of 3-gram instructions in the decompiled file has been collected for the resulting flow of instructions and their text sequences from the tree. Since the sequences of three consecutive instructions are too many to be used as features for classifiers, 400 sequences with the highest information gain rate were taken. The results obtained by the classifiers (k-nearest neighbors, decision tree, and support vector method) were better for the features derived from the control tree: the rate of correct responses is higher, and the false positive rate and false negative rate are lower.

Also, in [13] a moderate relationship between GCC compiler options and the frequencies of n-gram instructions was found, but the hypothesis was not tested due to the lack of datasets for analysis.

Instruction frequencies data was also applied to evaluate the efficiency of hardware resource utilization. For instance, the instruction set of four ISA was analyzed [14], and it was found that on average only 5-20% of all instructions were used, measuring instruction set utilization by the SPEC CPU 2006 benchmark application group. The applications were compiled using GCC and PCC with the option to generate assembly files instead of binary files on Ubuntu 10.04 LTS.

The authors of [15] analyzed the set of x86 instructions in Windows 7 for 32- and 64-bit applications and collected various statistics for them. For each instruction, information about its type, arguments, length, and addressing mode was collected. From this data, statistics on the instruction frequencies, register usage, and number of occurrences of different types of addressing were gathered. The authors assume that this information is useful for developing intermediate languages such as Java bytecode to minimize memory consumption.

Another research [16] performs static analysis of virtual machine disk images. An instruction crawler was implemented that looks through each disk image and counts how many times each unique instruction signature (UIS) appears in any executable file. In addition to static analysis, dynamic analysis was performed on a modified virtual machine because new instructions may be generated and executed at runtime. The analysis was performed for 32-bit machines running different Ubuntu and Windows versions from 1995 to 2012. It was discovered which instructions were not used or had ceased to be used over time. Based on the collected data, a tool was developed to remove old instructions without affecting backward compatibility. However, its source code was not published.

Finally, the authors of [4] explored the relative importance of instructions based on the number of their occurrences in packages and the popularity of these applications. The study was conducted for the Ubuntu 16.04 distribution and x86-64 architecture. For each package, the frequency of instructions in binary files and in called libraries is recursively counted using the readelf and objdump utilities. The authors suggest using the data they collect to measure the completeness of binary tools, since for many applications it is not necessary to support all instructions. They also claim that 55 packages covering all instructions are sufficient to verify binary tools.

Summing up, analysis of machine instruction usage can be successfully applied to a variety of problems. However, many authors do not provide source code and datasets. This makes the results of their studies not reproducible and prevents other researchers from using the data for new experiments. We take this into account and provide access to both the source code of the framework and the received data. As one could see, we could not find studies using data analysis on the appearance of various machine instructions for planning software migration to new hardware architectures or developing optimizing compilers. For this reason, this work highlights new ways of using static analysis of machine code.

4. Implementation

In this section, we describe the capabilities of our framework as of May 1, 2023.

4.1 Data collection

To collect data, a program was written that provides many configuration options for the needs of specific users, as well as a convenient command-line interface that allows one to gather all the necessary data with a single command. Its main functionality is to count the number of instructions in all programs in the specified folder and its subfolders and save the received data in a CSV table. This is achieved using the `readlink` and `objdump` utilities.

On different GNU/Linux distributions. In order for data collection to take place on different GNU/Linux distributions, regardless of which operating system is installed on the machine that starts the data collection process, the program described above is run in Docker containers. Docker images

of distributions are built according to dockerfiles stored in the repository. This approach allows to attain extensibility: to add a new distribution for scanning, it is enough to add the corresponding dockerfile, and to add a new package, one just needs to write its installation in the dockerfile.

Local data collection can take a lot of time and resources. To solve this problem, data collection starts automatically on the servers. At the moment, the ability to launch via GitHub Actions is used. This happens in two stages. First, Docker images are built according to dockerfiles and published in the repository [17] on DockerHub. If the dockerfile has not been modified since the last GitHub Actions workflow, the image is not rebuilt. This determines such an architecture. At the next stage, data is collected on all distributions concurrently: in each distribution, a Docker image is loaded from DockerHub, a Docker container is launched, and the program described above is run. The collected data is uploaded to a temporary storage in the cloud, from where it can be taken for analysis.

For a better understanding of how long it can take to collect data on GitHub Actions, we present *Table 1*. It contains measurements for Manjaro, Ubuntu and OpenSUSE distributions on x86-64 architecture with `firefox`, `chromium`, `kcachegrind`, and `vlc` packages installed. As the data collection time, we took the median value for 15 launches.

Table 1. Measurements for data collection process

Image	Size of image, GB	Data collection time, s	Size of obtained data, MB
Ubuntu	1.26	623	8.17
OpenSUSE	1.66	620	6.88
Manjaro	2.34	1192	9.61

On different platforms. The framework provides the ability to scan disk images (currently, in `.iso`, `.img`, and `.vmdk` formats), which allows one to collect data from different ISA. One can run a script to obtain data from a disk image that is already downloaded or scan the image by its URL. Compressed image processing is provided too (for now, in `.xz`, `.7z`, and `.bz2` formats). In addition, data from disk images by their URLs can be gathered automatically using GitHub Actions.

In order to give everyone the opportunity to explore the data, come up with new or improve existing analysis and visualization tools, we have uploaded the datasets obtained on April 1, 2023 to the repository [18].

4.2 Data analysis

To simplify the typical tasks of analyzing data on the use of machine instructions, a library of domain-oriented auxiliary functions has been implemented, which extends standard pandas library functions for working with tabular data and allows, for example, in the Jupyter Notebook environment to answer subject area questions in a few lines of code with the possibility of interactivity. For instance, using the framework, one can solve the following tasks.

- Find the top N most popular or rarest instructions.
- Divide instructions into some clusters (described in more detail in the following subsection).
- Figure out in which files some instructions (or their categories or groups) are used. In particular, it makes it possible to discover where vector extensions of instructions for speeding up encryption are applied.
- Plot interactive histograms of the distribution of instructions (or their categories or groups).
- Quickly get full information about the instruction (based on third-party documentation) and more.

An example of data analysis with a demonstration of some of the capabilities of the tool is presented in the repository. Particular attention was paid to writing detailed documentation. It is published on GitHub Pages [19] and is updated automatically when changes occur.

4.3 Splitting instructions into categories and groups

One of the most significant difficulties that one faces when analyzing the use of machine instructions is their large number. As already mentioned, some instructions, although they are different, perform essentially the same task. In addition, users frequently want to find out where not a specific instruction is used, but some cluster of them, for example, vector or encryption acceleration extensions. For these reasons, it is necessary for the framework to provide the possibility of dividing instructions into certain categories. This problem is indeed non-trivial, since many sources [20] offer descriptions of instructions without any division into larger units. Others [21], having the division of instructions into groups, do not include some important extensions.

At the moment, the framework provides an approach for solving the described problem for the x86-64 architecture. To accomplish this, a Python program was written that collects information from Linux Assembly libraries project [22], covering a fairly large number of instructions. We call the *category* of the instruction the section of the site on the left where it is included, and the *group* – its subsection in it. Thus, a two-level clustering of instructions is achieved. The program collects a description, category, and group for each instruction and stores the result in a json file, the data from which is then used to divide instructions into groups and categories during data analysis. This file is also placed in the repository so that everyone can use it.

As an example, *Fig. 1* shows part of some histogram before splitting instructions into categories and groups. *Fig. 2* shows a histogram of the total instruction category distribution for the same data. As one can see, the latter may be more convenient and clear for perception.

5. Evaluation

The proposed method of categorizing instructions for the x86-64 architecture, however, does not cover all possible instructions. So, on Manjaro, Ubuntu and OpenSUSE distributions, with `firefox`, `chromium`, `kcachegrind`, and `vlc` packages installed, the number of occurrences of non-covered instructions ranges from 7 to 10 percent. We refer them to the “Other” group and category.

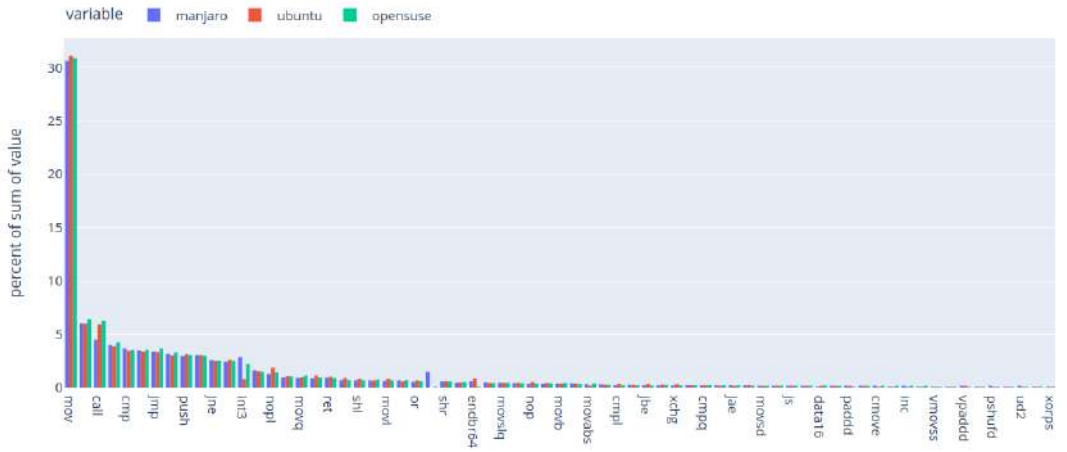


Fig. 1. Part of the histogram of the total instruction distribution before splitting into categories and groups

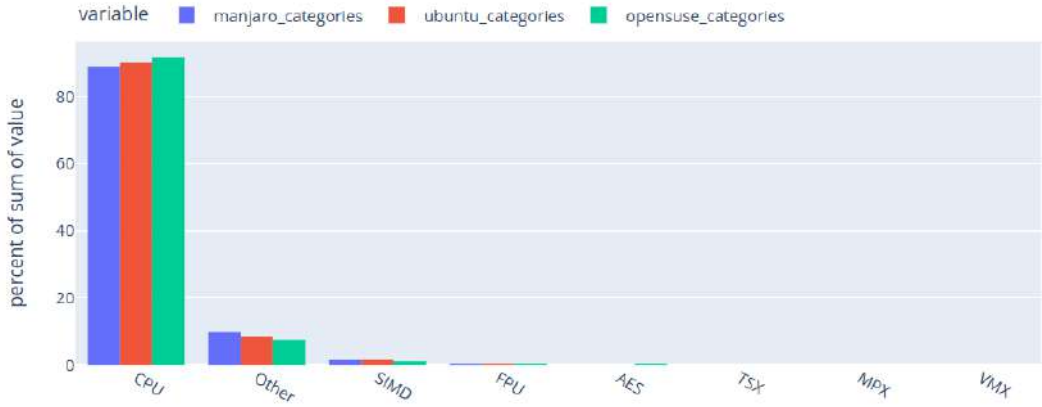


Fig. 2. Histogram of the total instruction category distribution

At the moment, the presented framework allows one to reproduce the main results of [4], which was discussed in Section 3, and differs from it in the following advantages.

1) The ability to update data whilst the results of [4] have not been updated for more than four years.
2) The ability to easily vary the analyzed applications by means of configuring building of distribution images in dockerfiles.
3) The ability to collect data from any GNU/Linux distribution.
4) The ability to collect data from any architecture.
5) Two-level clustering of instructions.
6) The ability to flexibly change data filters and visualization tools.

6. Future work

As mentioned in Section 1, it is often necessary to compare the use of certain groups of instructions, for example, vector or encryption acceleration extensions, in applications on different architectures. To do this automatically, it is necessary to divide the instructions into larger units for other architectures besides x86-64. We are planning to do it.

References

[1]. RISC-V International home page, Available at:: https://riscv.org/about/ (accessed: 01.05.2023).
[2]. RISC-V Alliance in Russia, Available at: https://riscv-alliance.ru/ (accessed: 01.05.2023).
[3]. Global News on High Performance Computing (HPC), Available at: https://www.hpcwire.com/2022/12/16/europe-to-dish-out-e270-millionto-build-risc-v-hardware-and-software/ (accessed: 01.05.2023).
[4]. Akshintala A., Jain B., Tsai C., Ferdman M., Porter D. X86-64 Instruction Usage among C/C++ Applications. Proceedings of The 12th ACM International Conference On Systems And Storage. pp. 68-79 (2019), DOI: 10.1145/3319647.3325833.
[5]. GitHub repository, Available at: https://github.com/DanilaPechenev/InstructionAnalysisFramework/tree/syrcose (accessed: 01.05.2023).
[6]. Kollara A. Opcode Frequency Based Malware Detection Using Hybrid Classifiers. National College of Ireland, 2020.
[7]. Bilar D. Opcodes as Predictor for Malware. Int. J. Electron. Secur. Digit. Forensic. 1, 156-168 (2007,1), DOI: 10.1504/IJESDF.2007.016865.
[8]. Baldwin J., Dehghantanha A. Leveraging support vector machine for opcode density based detection of crypto-ransomware. Cyber Threat Intelligence. pp. 107-136 (2018), DOI: 10.1007/978-3-319-73951-9 6.
[9]. Rad B., Masrom M., Ibrahim S. Opcodes histogram for classifying metamorphic portable executables malware. 2012 International Conference On E-Learning And E-Technologies In Education (ICEEE). pp. 209-213 (2012), DOI: 10.1109/ICeLeTE.2012.6333411.

[10]. Han K., Kang B., Im E. Malware Classification Using Instruction Frequencies. Proceedings Of The 2011 ACM Symposium On Research In Applied Computation. pp. 298-300 (2011), DOI: 10.1145/2103380.2103441.
[11]. Shabtai A., Moskovitch R., Feher C., Dolev S., Elovici Y. Detecting unknown malicious code by applying classification techniques on opcode patterns. Security Informatics. 1, 1-22 (2012).
[12]. Ding Y., Dai W., Yan S., Zhang Y. Control flow-based opcode behavior analysis for Malware detection. Computers & Security. 44 pp. 65-74 (2014), DOI: 10.1016/j.cose.2014.04.003.
[13]. Kenneth V. Opcode statistics for detecting compiler settings. University of Amsterdam, 2018.
[14]. Mutigwe C., Kinyua J., Aghdasi F. Instruction set usage analysis for application-specific systems design. Int'l Journal Of Information Technology And Computer Science. 7 (2013).
[15]. Ibrahim A., Abdelhalim M., Hussein H., Fahmy A. An Analysis of x86-64 Instruction Set for Optimization of System Softwares. International Journal Of Advanced Computer Science. 1, 152-162 (2011, 10).
[16]. Lopes B., Auler R., Ramos L., Borin E., Azevedo R. SHRINK: Reducing the ISA Complexity via Instruction Recycling. SIGARCH Comput. Archit. News. 43, 311-322 (2015,6), DOI: 10.1145/2872887.2750391.
[17]. DockerHub repository, Available at: https://hub.docker.com/repository/docker/danilapechenev/instructionanalysis/general (accessed: 01.05.2023).
[18]. Obtained datasets, Available at: https://github.com/DanilaPechenev/InstructionAnalysisFramework/tree/syrcose-data (accessed: 01.05.2023).
[19]. Framework documentation, Available at: https://danilapechenev.github.io/InstructionAnalysisFramework/ (accessed: 01.05.2023).
[20]. x86 and amd64 instruction reference, Available at: https://www.felixcloutier.com/x86/ (accessed: 01.05.2023).
[21]. x86 Opcode and Instruction Reference, Available at: http://ref.x86asm.net/geek.html (accessed: 01.05.2023).
[22]. x86-64 Instructions Set (Linux Assembly libraries project), Available at: https://linasm.sourceforge.net/docs/instructions/index.php (accessed: 01.05.2023).

Информация об авторах / Information about authors

Данила Евгеньевич ПЕЧЕНЕВ является студентом и исследователем Санкт-Петербургского государственного университета. Его научные интересы включают программную инженерию, анализ производительности приложений, статический анализ, эвристические алгоритмы.

Danila Evgenevich PECHENEV is a student and researcher at St. Petersburg State University. His research interests include software engineering, application performance analysis, static analysis, heuristic algorithms.

Яков Александрович КИРИЛЕНКО является руководителем лаборатории технологий программирования инфраструктурных решений Санкт-Петербургского государственного университета. Его научные интересы включают реинжиниринг программных комплексов, технологии программирования, оптимизацию программного обеспечения, архитектуру вычислительных систем.

Iakov Aleksandrovich KIRILENKO is the head of the Infrastructure Solutions Programming Technologies Laboratory at St. Petersburg State University. His research interests include software reengineering, programming technologies, software optimization, and architecture of computing systems.

Ольга Андреевна АФОНИНА – студентка и исследователь Санкт-Петербургского государственного университета. Научные интересы: программная инженерия и статический анализ.

Olga Andreevna AFONINA is a student and researcher at Saint Petersburg State University. Her research interests include software engineering and static analysis.

DOI: 10.15514/ISPRAS-2023-35(3)-13

Using Process Mining to Leverage the Development of a Family of Mobile Applications

L.A. Rezunik, ORCID: *0009-0000-9428-4718 <lrezunic@gmail.com>*
A.I. Perevoznikova, ORCID: *0009-0009-8248-8954 <alice.castiel1@gmail.com>*
D.V. Eremina, ORCID: *0009-0008-0653-5889 <dveremina@edu.hse.ru>*
A.A. Mitsyuk, ORCID: *0000-0003-2352-3384 <amitsyuk@hse.ru>*

HSE University,
11, Pokrovsky boulevard, Moscow, 109028, Russia.

Abstract. Enterprises often provide their services via a family of applications based on various platforms. Applications in such a family can behave differently. Their development processes can differ as well. Moreover, modern development processes are often complex and sometimes vague. This can lead to bugs, defects, and unwanted discrepancies in applications. In this paper, we show that process mining can be applied to leverage the development in such a case. Real-life models can be discovered and investigated by the developer teams in order to reveal differences in application behaviour, find bugs, and highlight inefficiencies. We consider datasets with event data of two types. Firstly, we analyse event logs of Android and iOS applications of the same product family. Secondly, we consider event data from working repositories of these applications. We show how by analysing such datasets, the real-life development process can be discovered. Besides, application event logs can help to find more and less severe bugs and unwanted behaviour.

Keywords: software process; software development; process mining; mobile application; software product family.

For citation: Rezunik L. A., Perevoznikova A. I., Eremina D. V., Mitsyuk A. A. Using Process Mining to Leverage the Development of a Family of Mobile Applications. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 171-186. DOI: 10.15514/ISPRAS-2023-35(3)-13.

Acknowledgements. This work is an output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE University).

Применение методов интеллектуального анализа процессов в ходе разработки семейства мобильных приложений

Л.А. Резуник, ORCID: 0009-0000-9428-4718 <lrezunic@gmail.com>
А.И. Перевозникова, ORCID: 0009-0009-8248-8954 <alice.castiel1@gmail.com>
Д.В. Еремина, ORCID: 0009-0008-0653-5889 <dveremina@edu.hse.ru>
А.А. Мицюк, ORCID: 0000-0003-2352-3384 <amitsyuk@hse.ru>

Национальный исследовательский университет «Высшая школа экономики»,
Россия, 109028, г. Москва, Покровский бул., 11, стр. 10.

Аннотация. Коммерческие предприятия часто предоставляют свои услуги с помощью семейства приложений, разработанных для работы на различных платформах. Приложения в таком семействе могут вести себя по-разному. Процессы их разработки также могут отличаться. Более того, современные процессы разработки часто сложны, а иногда и не вполне четко определены. Это может

приводить к ошибкам, дефектам и нежелательным отличиям в поведении приложений. В этой работе мы показываем, что методы интеллектуального анализа процессов могут применяться в ходе разработки такого рода семейства приложений. Команды разработчиков могут синтезировать и исследовать модели реального поведения приложений для выявления отличий в их поведении, поиска ошибок и выявления проблем производительности. В работе рассматриваются наборы данных двух типов. Во-первых, анализируются журналы событий приложений для платформ Android и iOS из одного и того же семейства программных продуктов. Во-вторых, рассматриваются событийные данные из рабочих репозиториев, в которых происходит разработка этих приложений. Показывается, как, анализируя такие наборы данных, можно выявить характеристики реального процесса разработки. Кроме того, анализ журналов событий самих приложений может помочь обнаружить более или менее серьезные ошибки, а также нежелательное поведение.

Ключевые слова: программные процессы; разработка программного обеспечения; интеллектуальный анализ процессов; мобильные приложения; семейства программных продуктов.

Для цитирования: Резуник Л. А., Перевозникова А. И., Еремина Д. В., Мицюк А. А. Применение методов интеллектуального анализа процессов в ходе разработки семейства мобильных приложений. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 171–186 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–13

Благодарности: Данная работа является результатом исследовательского проекта, реализованного в рамках Программы фундаментальных исследований НИУ ВШЭ.

1. Introduction

It is a common practice for business enterprises to provide their services via different user applications. We can, for example, use a web-application at our desktop and a mobile application when outdoors. Moreover, users have different mobile devices based on various technologies. All this leads to a family of applications that is developed and maintained by an enterprise to provide its services directly to potential users on their familiar platforms. Companies can develop members of such a software product family separately, one-by-one, or maintain a common software development environment. Combined with modern agile development approaches, all this leads to high variability, complexity, and sometimes vagueness of the development process. This, in turn, can lead to bugs and defects in software.

To our fortune, software applications on all platforms generate a large number of data records in the process of their functioning. Different types of data are present: user activity logs, error and system logs, debug information, communication logs, and other. We can use these records to discover the actual development process, find its inefficiencies and drawbacks. Moreover, an investigation of application's event logs can shed light on its structure and behaviour (see Section VIII). Process mining [1], [2] is a particular field providing us with tools which help to extract valuable information and insights out of raw event data.

In this paper, we consider two datasets (see Section IV) for a concrete family of mobile applications (see Section II for a system description). The first of these datasets contains event data from the repositories with source code. The second one has been obtained by recording logs of how users interact with the applications.

The main goal of this paper is twofold: (1) to show how we can reveal the real development process of a family of mobile applications using event data, (2) to provide the reader with the approach to find drawbacks and errors in both applications and their development process.

Our case study is conducted in accordance with the PM^2 methodology [3] for process mining projects. By analysing applications log data, companies aim to improve their business processes [4]. We show that this methodology can be successfully applied within the domain of software engineering with valuable outcomes.

2. System

The analysis was conducted for a family of mobile applications — HSE App X, which includes iOS and Android applications with the same functionality.

HSE App X as a whole is a client-server application used by students and staff of HSE University to interact with the university's systems. As it was mentioned, there are two client applications — for iOS and Android.

Taking as an example the iOS application, it can be seen how the client is built (the access to the project repository was granted by the developers).

The client contains several modules: a module for authorization functionality, a module for Apple Watch application, widgets, and the main module, which represents the iOS application (see *Fig. 1*).

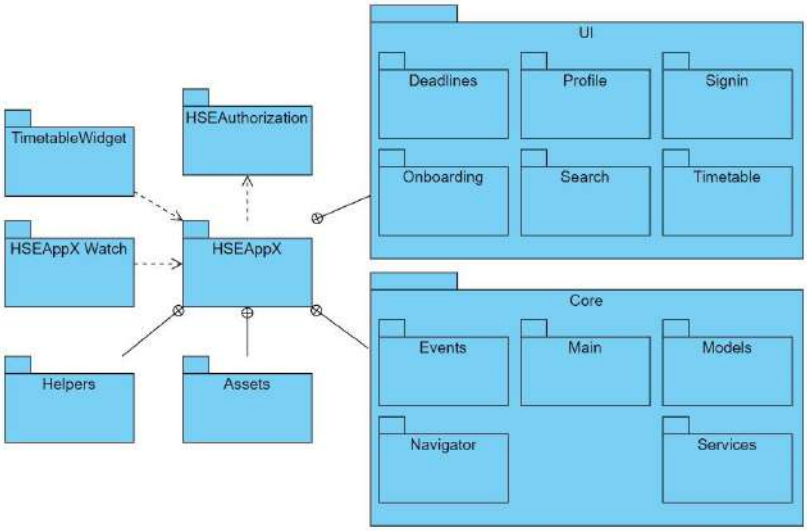


Fig. 1. Package diagram of the iOS client

The main module is organised into groups of files (packages): Core, UI, Assets, Helpers. A summary of each package is provided below.

Core — includes sub-packages which implement the logic for authorisation, system events, API calls. This package also contains the main entry point of the application.

UI — contains separate packages for each of the applications' screens.

Assets — package with various media files (images, sounds). Moreover, it stores all the fonts and colours used in the application along with localisation.

Helpers — holds helper classes, extensions for existing classes, mocks.

An example of a typical user scenario is viewing the timetable. What happens on application start:

1) All the classes that need to connect to the API are initialised.
2) The application sends requests to the API to get:
   a) the catalogues (e. g. all the HSE buildings) from the server;
   b) the user's notifications;
   c) information about the user;
   d) list of features available to a certain user.

Along with sending requests, the first screen of application loads (in parallel). As soon as the screen controller object is created, the authorisation token is verified and the request to get the user's

schedule is sent. After receiving a response, the application shows all the data on screen and saves it in cache.

Both Android and iOS applications are logging results of API calls, system events and user activity. Thus, the logs can be effectively used for analysis.

3. Research Tasks and Questions

In this paper, our goal is to perform the following tasks and to answer related research questions:

1) to build a process model for various user scenarios and determine if there are any "abnormal" (deviating from the norm) events. If there are any, try to explain them;
2) to analyze the process model for iOS and Android applications and find differences. If there are any, find out if they affect the performance and operation of the system, whether they need to be resolved;
3) to check if there are inconsistencies between the request body and the server response;
4) to identify the sequences within the process model that lead to error with the greatest frequency;
5) to analyze the data from the project repository and evaluate the team's work style;
6) to track which parts of the code (modules, files) have not been refactored/redesigned for the longest time and that are worth paying attention to.

4. Data

4.1 Data

In order to answer the research questions, two main sources of data were taken into consideration: logs of mobile applications and the project's repository.

Mobile applications log the user's actions, making it possible to extract files with a detailed history. They contain all the necessary information about the time and content of each request sent to the server and system logs as well. The developers are provided with different options to interact with debug mode of the application (it is hidden from regular users), which is helpful in obtaining suitable data for analysis. For these purposes it is necessary to clear previous session logs, so it becomes possible to focus on the particular user's interactions.

Log files collected for particular use cases of the application were parsed by found patterns and then aggregated using common Python libraries. As a result, the data contained structured information about requests' URL, body and response with corresponding timestamps.

The repository stores all the project members' activity: commits, merge requests, issues, etc. This data can be effectively used to analyze the processes within the team. Considering the research questions, it was identified that commits hold the most information. Thus, all the commits stored in the GitLab repository were extracted into a single .csv file. For data retrieval, a few scripts were written using the Java library GitLab4J to make the process of working with GitLab's API more straightforward.

The resulting data file contains all the needed information about the commits: id, author's name, title of commit, all the changes (file paths), and timestamp, which corresponds to the exact date and time the commit was made locally (before push).

4.2 Data Preprocessing

The previous section described the process of collecting the data from the project's repository, resulting in only the necessary data for analysis remaining, including commit id, date and time of commit creation, author's name, and all the names of changed files.

In addition, it was decided that commits should be also grouped in some way, because the number of individual commits was too large to be efficiently processed by some process mining tools. Knowing that the project was managed using Agile methodology, sprint number was added to each of the commits (sprint is a 2 week long time interval). Thus, each of the commits was assigned a sprint it was created in. This also improved the dataset, because not only time of creation of a single commit could be used for analysis, but also a group of commits.

Besides, the logs of iOS and Android applications were collected separately, after which they were converted into a format suitable for processing. After processing the data, it acquired the following form: session id, date and time of the request, request URL (without parameters), response status, duration of the request, message (see *Fig. 2*).

session	timestamp	url	status	elapsed	message
0	2023-01-14 18:03:4	https://api.hseapp.ru/v2/dump/favourites/me	200	0.109	
0	2023-01-14 18:03:4	https://api.hseapp.ru/toggles	200	1.085	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v2/notifications/feed	200	0.099	
0	2023-01-14 18:03:4	https://api.hseapp.ru/fcm/devices	200	0.081	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v3/ruz/lessons	200	0.987	
0	2023-01-14 18:03:4	https://api.hseapp.ru/banners	200	0.222	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v3/ruz/lessons	200	0.989	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v2/deadlines/invites	200	0.222	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v2/deadlines	200	0.352	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v2/deadlines/groups	200	0.236	
0	2023-01-14 18:03:4	https://api.hseapp.ru/v2/deadlines	200	0.348	
0	2023-01-14 18:03:5	https://api.hseapp.ru/v2/deadlines/disciplines	200	0.845	
0	2023-01-14 18:04:4	https://api.hseapp.ru/v2/deadlines	200	0.21	
0	2023-01-14 18:04:5	https://api.hseapp.ru/v2/deadlines/639656523b560777db366c9f/con	200	0.146	
0	2023-01-14 18:05:0	https://api.hseapp.ru/v2/deadlines/6346eca8369372faabc46a94/com	200	0.128	
0	2023-01-14 18:05:1	https://api.hseapp.ru/v2/deadlines/6346d3ab2deb6df1d013eac1/con	200	0.109	
0	2023-01-14 18:05:1	https://api.hseapp.ru/v2/deadlines/63c2c48cdaec719d41440b6e/con	200	0.114	
1	2023-01-14 17:59:2	https://api.hseapp.ru/v2/dump/favourites/me	200	0.123	
1	2023-01-14 17:59:2	https://api.hseapp.ru/v2/notifications/feed	200	0.068	
1	2023-01-14 17:59:2	https://api.hseapp.ru/toggles	200	1.46	

Fig. 2. Example of a dataset collected from the Android application

At the same time, both the request and the response are logged in the Android application, so it was necessary to remove duplicates from the file. Also, in the Android application, in case of incorrect requests, incorrect dates appeared that had to be processed. To determine the case id, the session number was added (a separate application launch), and the date and time format was also changed to fit the mining algorithms provided by mining tools. Additional parameters were removed from the URLs that would interfere with building a graph due to the presence of too specific information.

5. Development Process Analysis

5.1 Analysis

The dataset from the previous section, which was formed using repository data, was applied to interpret the processes among the developers. Analysis of the data was conducted using ProM [5], a program specifically developed for process analysis. It provides a big tool set, which was used throughout all research.

Firstly, the team workflow was analysed by generating dotted chart diagrams with different display options. For example, by putting the value of the commit creation time (since the start of the week) on the X axis and commit id on the Y axis, a chart representation of developer's working schedule was built (see *Fig. 3*).

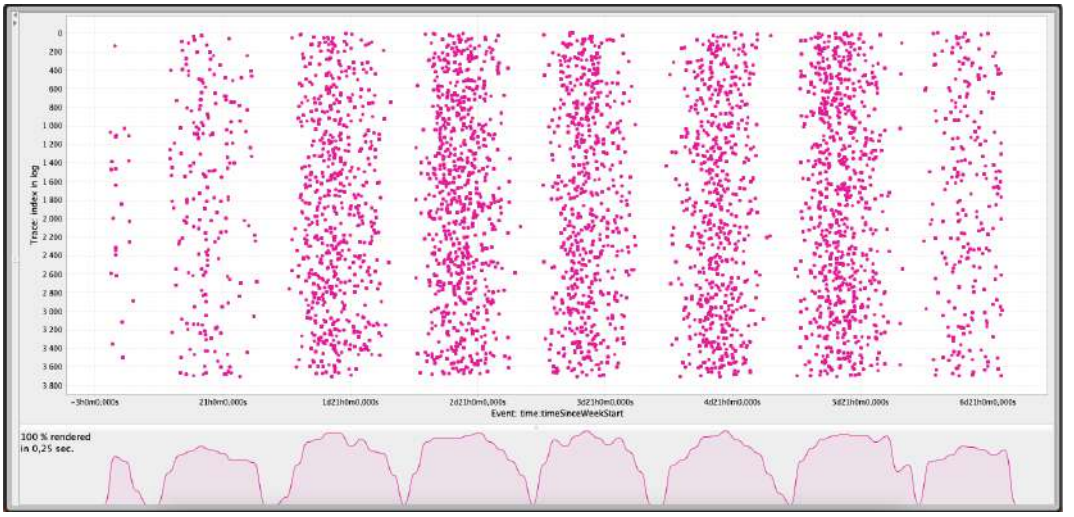


Fig. 3. Dotted chart that shows developers' working schedule

Moreover, the time of creation of all commits was analysed. The constructed dotted chart gave representation of developers' individual working hours.

This method of analysis was also used to discover project members' involvement in the development process. It can be seen through the dependency between changes in project modules and the author of the commit. The diagram (see *Fig. 4*) illustrates the division of duties among the developers, and shows which pieces of code can only be changed by a particular person.

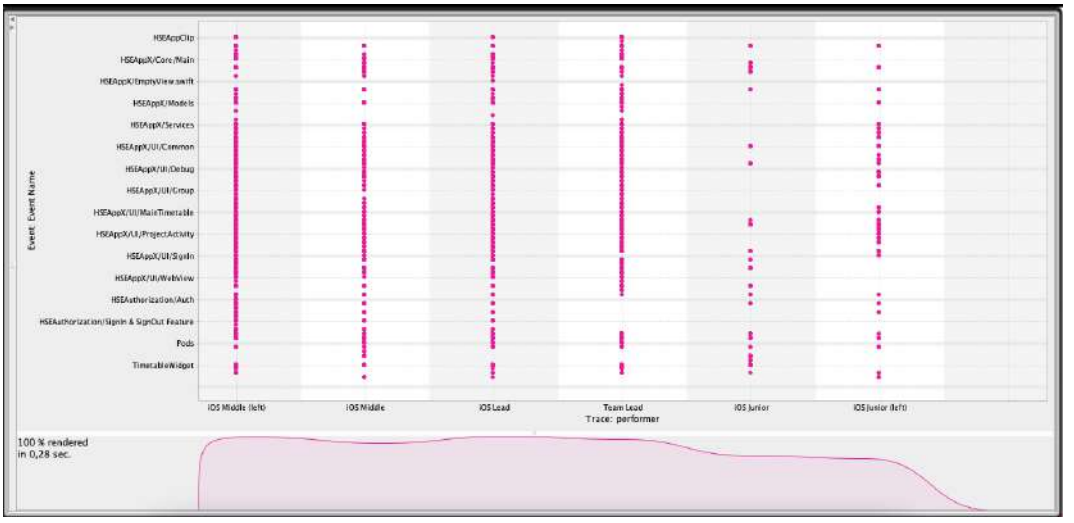


Fig. 4. Dotted chart that shows the commits submitted by developers for certain modules

In addition to the dotted chart diagrams, which resemble statistical research methods, some models were synthesised by applying the inductive mining algorithm (using the Inductive Miner [6] inside ProM). These models were used to see the transfer of work (if it exists) between the developers. Instead of inspecting individual commits, groups of commits were taken into consideration (each group was represented by sprint number).

The inductive miner algorithm requires a dataset with 2 columns selected. The first column represents case id, which identifies a single trace of the process (in this case it is one sprint — 2

weeks). The second selected value type — activity, which basically is an event in the trace (name of commit's author was used).

Thus, the resulting model (see *Fig. 5*) shows the generalised behaviour of the team members during one sprint. Conclusions on this model will be given in the next section of the document.

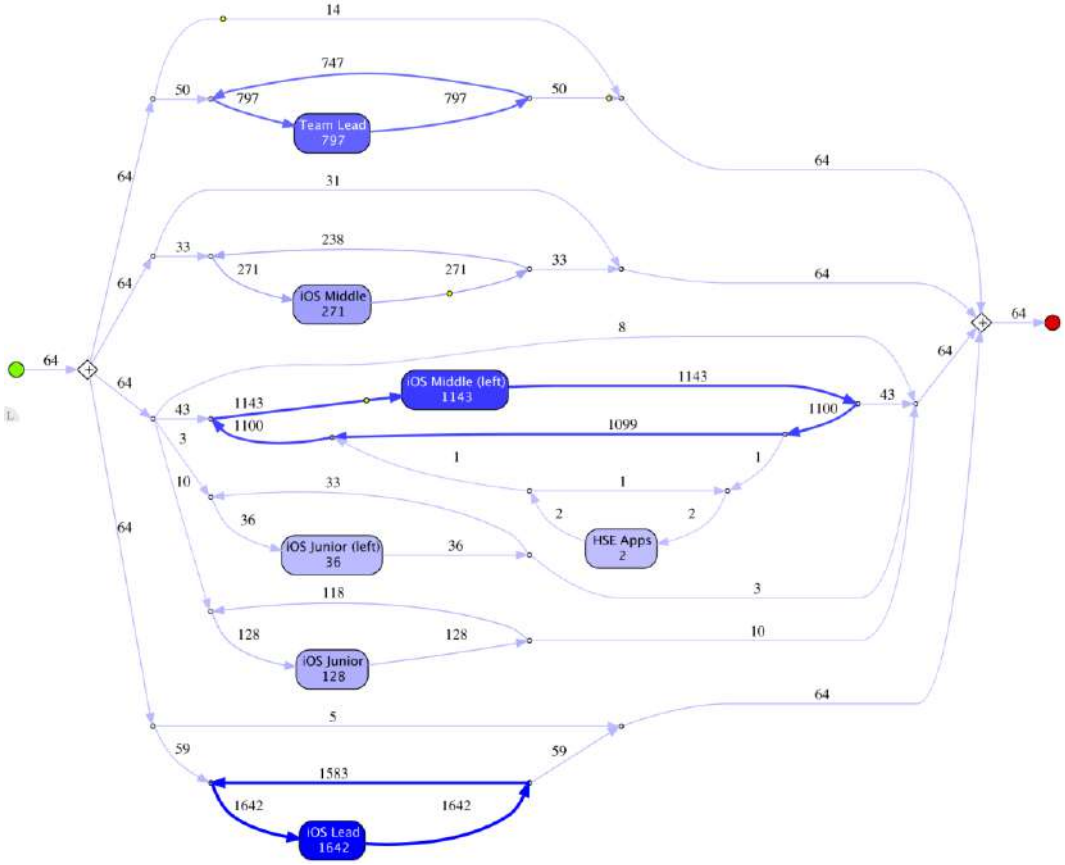


Fig. 5. Model depicting activity of all developers during one sprint

Another approach for viewing data, which was used in the research — generating a skeleton of the event journal. This model can give a clear understanding of the dependencies between the execution of events in one trace of the process. If the trace is represented by a single commit and the events in the trace are file changes, the resulting model can depict which files do not ever change together in a commit or vice versa.

5.2 Main Findings

After analysing the generated dotted chart diagrams, it was noticed that the team does not have a strict working schedule, commits in the repository are made every day of the week. However, there is a tendency among the developers to leave the weekends to themselves, that is why there is less activity on Sunday and Saturday.

Moreover, the working hours are not established for any of the team members. Distribution of the commits covers almost all the area of a one-day timeline. The developers can make changes to the project even in the early morning, although the peak activity is in the daytime.

The dotted chart diagram made it possible to assess in more detail the degree of familiarity of developers with individual modules of the project. For example, it was noticed that the person who

was no longer on the team was initially working on the authorisation module, and the other developers' tasks only included small changes and refactoring of that code.

By applying the inductive miner algorithm (described in the previous section) a model was produced, which depicts the order in which the developers make commits in a sprint. As it can be seen there is no dependency between the developers. The team members work in parallel, there are situations when a person does not make a single commit per sprint, but there is no certain sequence of work transfer between developers. This also shows that the tasks are mostly independent of each other.

It was also discovered that some developers never co-existed in the team: iOS Middle, both of the iOS Juniors and HSE Apps, which is another account run by the project manager.

The Log Skeleton Visualiser [7] was applied to discover relations between packages in a commit (if there are such relations). Thus, the case id for the mining algorithm was represented by the id of a single commit and the activity was represented by the changed file.

It was noticed that some changes never co-occur in a commit. Developers try to keep their commit history clean, thus, it is not favourable to make changes in completely separate modules at once.

Taking for example Apple Watch module and Widget module, it is true that project members try to change files in those modules separately (see *Fig. 6*).

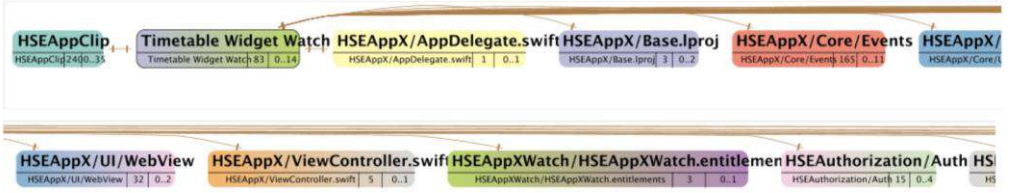


Fig. 6. Fragment of the log skeleton model depicting relations between file changes

While looking at the statistics overview for all the commits, it was noticed that the number of changed files in one commit has a strong variation: from 1 changed file up to 693 files.

It is strongly advised to not have too many file changes in a single commit, because the code needs to go through a more in-depth code review. The developer in that case cannot keep track of all the changes, with a high probability such code may be bugged.

6. Application Behaviour Analysis

6.1 Analysis

For each dataset one of the application usage scenarios from the following list was reproduced:

1) *Viewing a personal timetable*. The user gets to the main screen of the application, looks at his timetable, and then opens the page of a certain lecture (or another type of activity) to view information about it.
2) *Searching for a person's timetable*. The user goes to the search section from the main screen, searches for the person, looks through his timetable.
3) *Searching for free classrooms*. The user goes to the services section from the main screen, configures the parameters for searching for classroom (building, date, time), after which he receives a list of available classrooms and views the schedule of a specific one.
4) *Viewing the grade book*. The user goes to the profile screen from the main screen, where he selects his grade book, a year of study, a discipline for which statistics should be viewed. After that, he returns to the screen with the grade, and clicks on the cell with his rating to view the full rating list.

5) *Deadline monitoring*. The user goes to the deadlines section from the main screen, creates a new personal deadline, sets the necessary parameters (discipline, title, description, participants, time), saves it and marks it as completed. Similar sequence of actions should be performed in case when the user creates a new group to add a common deadline.

To compare the conformance for different scenarios, Log Skeleton models were built. Using the example of one scenario, it can be seen that the models are generally similar to each other, they send requests to the same URLs, and in full use case scenario go all the way to the certain disciplines (in case of the first scenario).

At the same time, separate models were built that take into account the responses and errors returned by the requests (see *Fig. 7*).

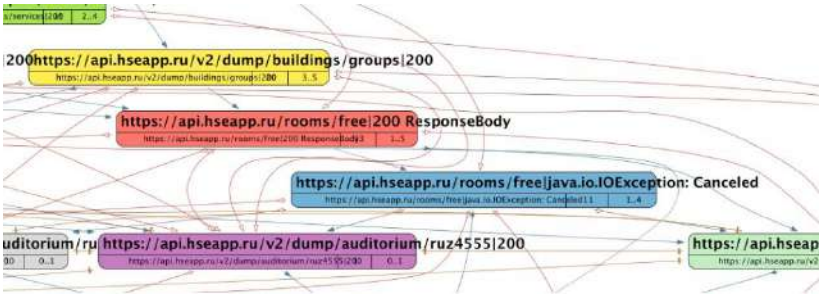


Fig. 7. Fragment of the log skeleton visualisation of the schedule viewing scenario in an iOS application with detected errors

Such anomalies were found in almost all scenarios, which made it possible to analyse the causes of such errors. For example, the Android application does send requests that are cancelled afterwards to avoid "race condition" in the application, which confirms the detected problem.

In addition, Petri nets were built for each process by applying Inductive Miner. Using the example of the "Timetable View" scenario, there is a noticeable discrepancy in its execution of the passage from the main screen to the necessary target (the screen of a specific discipline in the timetable). However, it is clear that applications send similar requests and reach similar endpoints (see *Fig. 8*).

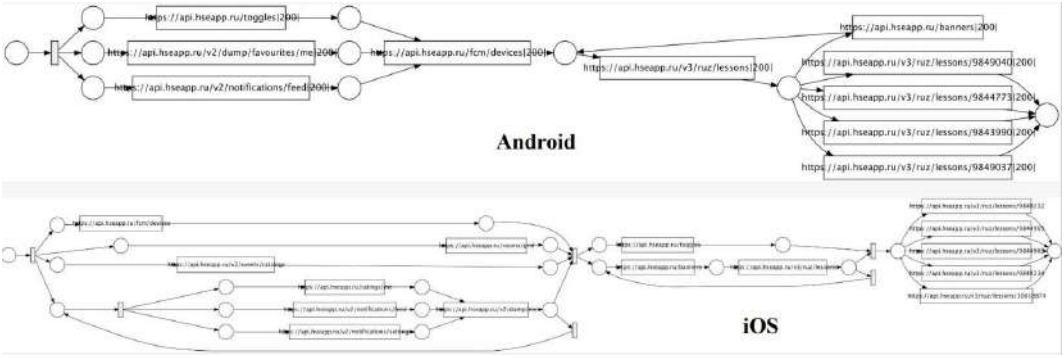


Fig. 8. Petri nets depicting the process of viewing the schedule using Android and iOS applications

6.2 Main Findings

A research study of mobile application logs was conducted to compare the operations of iOS and Android mobile applications, with a focus on identifying differences and unexpected behaviour patterns. The dataset of mobile application logs was analysed with the aim of uncovering insights that could help the business and the developers optimise their mobile apps for each platform. In this section, the main findings of this study are presented, highlighting the key differences and similarities between iOS and Android mobile applications.

First of all, in the study was used the Log Visualiser and summaries for each use-case to gain insights into the types of server requests being made on each platform. The Log Visualiser provided a graphical representation of the logs, while the summaries allowed for a more detailed analysis of the data. The analysis of the logs revealed that there were similar server requests being made on both platforms, with matching URLs. However, the frequency of these requests varied depending on the platform, due to the involvement of different screens. Despite these differences, the behaviour of the mobile applications on both iOS and Android platforms was found to be quite similar. Overall, this part of research contributes to a better understanding of high-level application behaviour, while next mining approaches helped to find out more detailed information.

After summaries analysis, several models were built to discover any differences in performed activities and their drawbacks. The Petri Nets were used to visualise the traces of the mobile applications. The use of such models in this study was beneficial as it allowed us to identify potential bottlenecks, because requests were the same at first sight and had no significant findings. However, there were some unexpected steps in traces. This led to usage of the Log Skeleton Visualiser in ProM, which provided a detailed graph of requests and their corresponding responses, allowing for a more comprehensive analysis of the mobile application logs. One of the key findings of this research was that the Android application logs graphs almost always contained an unexpected node with the error message '`java.io.Exception: Canceled`'. This finding prompted us to take a closer look at the processes involved in the Android application logs. Upon closer examination, it was discovered that there were confirmed issues with sending requests and cancelling them due to race conditions, which were likely contributing to the found type of error in the Android application logs graphs. Moreover, a repeating error in some log traces of the iOS application was found. It was caused by incorrect parsing of the data that was coming from the server.

This research highlighted the need to address the issues in the Android application to improve its performance and reliability. Such analysis can help developers to optimise their software and prevent similar issues from arising in the future.

7. Discussion

During the research of application processes, Android and iOS logs were collected and prepared for analysis, as well as a representation of individual scenarios in the form of models obtained from them. To do this, it was necessary to clear the logs of useless information, parse the needed parts, bring them to a tabular form and select columns for analysis. This made it possible to track the behaviour on various platforms, to identify patterns and anomalies in the requests, which should lead to the same result.

7.1 Answers to Research Questions

During the analysis of diagrams, models and the general report in relation to a set of different scenarios, it was found out that there are no critical discrepancies regarding the requests sent, since the requests URLs, their number, as well as the overall behaviour on the traces coincided. Thus, it can be assumed that there are no abnormal events and discrepancies in the general case. This answers the first research question.

To answer the second research question a more detailed analysis of the logs collected in one scenario was conducted. It allowed us to identify erroneous events that are often found only in the Android application. On Log Skeleton models, it was noticeable that an event with a request cancellation error appears in the response. There were no dependencies on specific scenarios, but it was clear that such a response was returned as a result of sending a duplicate request that had already left the application. Serialisation errors were noticed for the iOS application, which did not lead to further deviations, but occurred in several traces.

It was also found that the sequences which are more prone to errors are those which include search requests (for Android). For iOS such obvious sequences were not found. There also weren't any

inconsistencies between the response data and application requests. Thus, the research questions 4 and 3 were also answered.

As a result of analysing the data from the repository, it became possible to understand the team's work style (the daily routine, working days of the week, methodology of development, quality of commits — size, which parts of the code were affected).

As it turned out, the developers' work schedule is not strictly set, each team member independently determines when it is more convenient for him to work. It can be noticed that developers commit to the repository less often on Saturday and Sunday. That is, programmers, even if they are not limited in the choice of working hours, decide not to work on weekends. In the same way, it is noticeable that commits are made at completely different times.

Project participants work on tasks in parallel with each other and everyone's familiarity with the project is approximately on the same level. Everyone has worked with almost every file in the repository at some point in time. The only exception is the junior developer, which is logical, given his level.

It was also possible to notice that commits are carried out in all modules of the project so far. It can be assumed that the tasks relate to different parts of the project and they can be issued to different developers. This approach to working on software products is also popular in large companies — when new functionality is gradually added to the product over time and when needed. That suggests any of the Agile methodologies is used in the development process (or no methodology is used at all, and programmers work the way they are used to). This concludes the answer of the fifth question of this research.

It was also possible to identify the parts of the project in which commits have been carried out the least recently (the last research question). The informativeness of such a list is a little doubtful, since there are several auxiliary files (for example, SberPaySDK). There are also many references

to the authorisation module and to the main module of the project. With the help of additional tools, it was possible to identify several files where commits have not been performed for a long time — these are files related to network interaction and some basic files. However, this information is not accurate enough, since the analysis included large commits affecting almost all project files.

7.2 Open Questions and Problems

Among the open questions regarding the various versions of the application, is an in-depth comparison of the body of the response to the request with the current status. To do this, it is necessary to improve parsing, explore examples of responses and such cases in logs, on the basis of which it will be possible to build new models. The detection of such discrepancies will improve the responses from the service, which will affect their processing in applications. It would be better to test the application work on a large number of scenarios and with the participation of real users.

8. Related Works

This section is a brief review of the field of mining software data. Process mining considers software as a research object for more than fifteen years [8], [9], and a lot of contributions have been made in this field. Most of them can be grouped into two major classes depending on what processes are considered:

- software behaviour,
- software (development, maintenance etc.) process.

Let us consider papers of both these classes. In this paper, we begin with the analysis of a development process. Therefore, in this section, we will follow the same approach.

The problem of team work assessment is crucial in the domain [10]. Indeed, process mining can be used to evaluate process performance. Software development processes can be evaluated more or less in the same way as other business processes. To achieve more detailed results, mining of event

logs can be combined with more conventional approaches like surveys and expert evaluation [11].

Lightweight development approaches are a target for mining due to their agility and non-linear nature [12]. Marques et at. [13] evaluates programmer's activities within agile development teams. The aim is to find what Scrum disciplines and practices can be revealed and checked using event logs from a case-handling system. Interestingly, some of the practices can be discovered using very basic techniques.

Mining event logs of individual programmers is another emerging sub-domain [14], [15]. Process mining techniques can be used, for example, to assist and fine-tune the learning process of novice developers [16]. Supplementary data from repositories of student projects can be investigated as well [17]. Such an analysis gives many ideas of how to improve and evolve existing programming courses in more interactive learning facilities. Security training of programmers and users can benefit from user log mining no less than other sub-domains [18], [19].

Let us now consider applications of process mining to software behaviour analysis.

Process analysis methods can be applied to very low-level technical problems. For example, Wakup and Desel [20] considered how logs of an application with active TCP/IP information interchange can be analysed using process mining. Authors constructed models of client-server communication with both sides clearly separable.

Leemans et al. proposed [21] and later developed [22] a methodology to analyse event logs of software systems. The goal is to reverse engineer a (possibly, legacy) system to reveal and grasp its behavioural characteristics. The methodology combines a general approach to software process analysis (similar to P M 2 [3]) with very technical tools [23]. Leemans et al. even constructed a ProM plug-in — Statechart Workbench [24] — that can support users of this approach.

In 2015, Shershakov and Rubin published a paper [25] on how to analyse real-life software executions and what can be discovered from event logs of such executions. Later, the team of collaborators of Sergey Shershakov developed several techniques to synthesise UML activity diagrams and other types of visual model for service-oriented and component-based systems [26], [27].

The same component-based systems have been analysed by Liu with his co-authors [28–30] In a series of papers, they worked on a problem of identifying communicating components and interfaces through which components communicate. Separate services in an enterprise-scale service-oriented system can be discovered based on event log analysis as well [31].

Process analysis approaches can be valuable in more specific sub-domains of software engineering. Software reliability can be assessed based on process mining techniques. A lot of approaches reviewed by Macak et al. [32]. Interestingly, even the evolution of highly-distributed systems like block-chain applications can be successfully revealed using process analysis approaches [33].

Mobile applications can also be the object for process analysis. Process mining is a popular research topic in various fields related to user applications, such as mobile games [34] and others. Usually, the goal of the research projects is to solve some common problem and find various (anti-)patterns in user behaviour. Sometimes, researchers are trying to answer business-related questions for particular mobile applications [35]. Log analysis for a specific platform was also mentioned in literature [36], but without the goal to compare behaviour of several applications for different platforms.

From this concise literature review, we conclude that as software generates a lot of data, this data can be analysed with valuable outcomes for developers, designers, and users. The field of software process analysis develops actively. Methods applied in our paper are in line with the state-of-the-art approaches in process mining. The object of our investigation — a family of mobile applications — is new for the field.

9. Conclusion

In this paper, we presented the results of a case study. We applied process mining to analyse datasets containing event data recorded within the development process of mobile applications. Usually, mobile applications are developed and maintained as a family because enterprises want to provide their services on various platforms. In the paper, we show that different applications of the same family can behave in a variety of ways. Besides, their development processes can differ as well. Process mining allows for discovering real process models which can be easily investigated by developer teams in order to reveal unwanted discrepancies, find bugs, and highlight inefficiencies.

References

[1]. van der Aalst W. M. P. Data Science in Action – Second Edition. Springer, 2016. 467 p.
[2]. van der Aalst W. M. P., Carmona J. Process Mining Handbook. Springer, vol. 448, 2022. 503 p.
[3]. van Eck M. L., Lu X., Leemans S. J. J., van der Aalst W. M. P. PM^2: A process mining project methodology. CAiSE. Springer, vol. 9097, 2015, pp. 297-313.
[4]. van der Aalst W. M. P., Weijters T., Maruster L. Workflow Mining: Discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering, vol. 16, n. 9, 2004, pp. 1128-1142.
[5]. Verbeek E., Buijs J. C. A. M., van Dongen B. F., van der Aalst W. M. P. ProM 6: The process mining toolkit. CEUR Workshop Proceedings, vol. 615, 2010.
[6]. Leemans S. J. J. Robust Process Mining with Guarantees – Process Discovery, Conformance Checking and Enhancement. Springer, 2022. 467 p.
[7]. Verbeek H. M. W. The Log Skeleton Visualizer in ProM 6.9. Int. J. Softw. Tools Technol. Transf., vol. 24, n. 4, 2022, pp. 549-561.
[8]. Rubin V. A., Günther C. W., W. M. P. van der Aalst, Kindler E., B. F. van Dongen, Schäfer W. Process mining framework for software processes. ICSP: Lecture Notes in Computer Science. Springer, vol. 4470, 2007, pp. 169-181.
[9]. Rubin V. A., Mitsyuk A. A., Lomazova I. A., W. M. P. van der Aalst. Process mining can be applied to software too! ESEM. ACM, 2014, pp. 57:1-57:8.
[10]. Caldeira J., F. B. e Abreu, J. P. dos Reis, Cardoso J. Assessing software development teams' efficiency using process mining. ICPM. IEEE, 2019, pp. 65-72.
[11]. Vavpotic D., Bala S., Mendling J., Hovelja T. Software process evaluation from user perceptions and log data. J. Softw. Evol. Process, vol. 34, n. 4, 2022.
[12]. Rubin V. A., Lomazova I. A., W. M. P. van der Aalst. Agile development with software process mining. ICSSP. ACM, 2014, pp. 70-74.
[13]. Marques R., M. M. da Silva, Ferreira D. R. Assessing agile software development processes with process mining: A case study. CBI. IEEE Computer Society, 2018, pp. 109-118.
[14]. Ioannou C., Burattin A., Weber B. Mining developers' workflows from IDE usage. CAiSE Workshops: Lecture Notes in Business Information Processing, vol.316. Springer, 2018, pp. 167-179.
[15]. Ardimento P., Bernardi M. L., Cimitile M., Maggi F. M. Evaluating coding behavior in software development processes: a process mining approach. ICSSP. IEEE / ACM, 2019, pp. 84-93.
[16]. Ardimento P., Bernardi M. L., Cimitile M., Ruvo G. D. Learning analytics to improve coding abilities: a fuzzy-based process mining approach. FUZZ-IEEE. IEEE, 2019, pp. 1-7.
[17]. Macák M., Kruzelova D., Chren S., Buhnova B. Using process mining for git log analysis of projects in a software development course. Educ. Inf. Tecnol., vol. 26, n. 5, pp. 5939-5969, 2021.
[18]. Macák M., Oslejsek R., Buhnova B. Process mining analysis of puzzle-based cybersecurity training. ITiCSE. ACM, 2022, pp. 449-455.
[19]. Macák M., Oslejsek R., Buhnova B. Applying process discovery to cybersecurity training: An experience report. EuroS&P Workshops. IEEE, 2022, pp. 394-402.
[20]. Wakup C., Desel J. Analyzing a TCP/IP-protocol with process mining techniques. Business Process Management Workshops: Lecture Notes in Business Information Processing, vol. 202. Springer, 2014, pp. 353-364.

[21]. Leemans M., van der Aalst W. M. P. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. MoDELS. IEEE Computer Society, 2015, pp. 44-53.

[22]. Leemans M., van der Aalst W. M. P., van den Brand M. G. J., Schifferers R. R. H., Lensink L. Software process analysis methodology – A methodology based on lessons learned in embracing legacy software. ICSME. IEEE Computer Society, 2018, pp. 665-674.

[23]. Leemans M., van der Aalst W. M. P., van den Brand M. G. J. Recursion aware modeling and discovery for hierarchical software event log analysis (extended). CoRR, vol. abs/1710.09323, 2017.

[24]. Leemans M., van der Aalst W. M. P., van den Brand M. G. J. The statechart workbench: Enabling scalable software event log analysis using process mining. SANER. IEEE Computer Society, 2018, pp. 502-506.

[25]. Shershakov S. A., Rubin V. A. System runs analysis with process mining. Modeling and Analysis of Information Systems, vol. 22, n. 6, pp. 818-833, 2015.

[26]. Davydova K. V., Shershakov S. A. Mining hybrid uml models from event logs of soa systems. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), vol. 29, n. 4, pp. 155-174, 2018.

[27]. Zubkova N. S., Shershakov S. A. Method for building uml activity diagrams from event logs. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), vol. 31, n. 4, pp. 139-150, 2019.

[28]. Liu C., van Dongen B. F., Assy N., van der Aalst W. M. P. Component behavior discovery from software execution data. SSCI. IEEE, 2016, p. 1-8.

[29]. Liu C., van Dongen B. F., Assy N., van der Aalst W. M. P. Component interface identification and behavioral model discovery from software execution data.

[30]. Liu C., van Dongen B. F., Assy N., van der Aalst W. M. P. A general framework to identify software components from execution data. ENASE. SciTePress, 2019, pp. 234-241.

[31]. Alwis A. A. C. D., Barros A., Polyvyanyy A., Fidge C. J. Function-splitting heuristics for discovery of microservices in enterprise systems. ICSOC: Lecture Notes in Computer Science, vol. 11236. Springer, 2018, pp. 37-53.

[32]. Macák M., Daubner L., Sani M. F., Buhnova B. Process mining usage in cybersecurity and software reliability analysis: A systematic literature review. Array, vol. 13, pp. 100120, 2022.

[33]. Müller M., Ruppel P. Process mining for decentralized applications. DAPPCON. IEEE, 2019, pp. 164-169.

[34]. Kwon H., Kim D. A method for churn analysis of new users of mobile games using process mining. ICIC Express Letters, vol. 7, n. 8, 2016.

[35]. Kim S., Kim D. Analyzing mobile application logs using process mining techniques: An application to online bookstores. ICIC Express Letters, vol. 9, n. 6, 2013.

[36]. Park Y. C. B., Cho I., Lee W. A log analysis of smartphone application usage: Focusing on domestic iphone users. Journal of the HCI Society of Korea, vol. 2011, n. 1, 2011.

Информация об авторах / Information about authors

Людмила Александровна РЕЗУНИК – бакалавр программной инженерии, НИУ ВШЭ. Сфера научных интересов: разработка мобильных приложений, архитектура программного обеспечения, анализ процессов разработки программного обеспечения.

Lyudmila Alexandrovna REZUNIK – Bachelor of Software Engineering, HSE. Research interests: mobile application development, software architecture, software process mining.

Алиса Игоревна ПЕРЕВОЗНИКОВА – бакалавр программной инженерии, НИУ ВШЭ. Сфера научных интересов: разработка мобильных приложений, web-разработка, архитектура программного обеспечения.

Alisa Igorevna PEREVOZNIKOVA – Bachelor of Software Engineering, HSE University. Research interests: mobile application development, web development, software architecture.

Дарья Валерьевна ЕРЕМИНА – бакалавр программной инженерии, НИУ ВШЭ. Сфера научных интересов: бэкенд разработка, архитектура программного обеспечения.

Daria Valerievna EREMINA – Bachelor of Software Engineering, HSE University. Research interests: backend development, software architecture.

Алексей Александрович МИЦЮК – кандидат компьютерных наук, доцент, старший научный сотрудник научно-учебной лаборатории процессно-ориентированных информационных систем (НУЛ ПОИС) факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: интеллектуальный анализ процессов, информационные системы, архитектура программного обеспечения, сети Петри.

Alexey Alexandrovich MITSYUK – PhD in Computer Science, Associate Professor, Senior Research Fellow at the Laboratory of Process-Aware Information Systems (PAIS Lab) of the Faculty of Computer Science at the HSE University. Research interests: process mining, information systems, software architecture, Petri nets.

DOI: 10.15514/ISPRAS-2023-35(3)-14

Predicate Abstraction Refinement in Thread-Modular Analysis

V.P. Rudenchik, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>
P.S. Andrianov, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>

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

Abstract. Thread-modular approach over predicate abstraction is an efficient technique for software verification of complicated real-world source code. One of the main problems in the technique is a predicate abstraction refinement in a multithreaded case. A default predicate refiner considers only a path related to one thread, and does not refine the thread-modular environment. For instance, if we have applied an effect from the second thread to the current one, then the path in the second thread to the applied effect is not refined. Our goal was to develop a more precise refinement procedure, reusing a default predicate refiner to refine both: a path in a current thread and a path to an effect in the environment. The idea is to construct a joined boolean formula from these two paths. Since some variables may be common, a key challenge is to correctly rename and equate variables in two parts of the formula to accurately represent the way threads interact. It is essential to get reliable predicates that can potentially prove spuriousness of the path.
The proposed approach is implemented on top of CPAchecker framework. It is evaluated on standard SV-COMP benchmark set, and the results show some benefit. Evaluation on the real-world software does not demonstrate significant accuracy increase, as the described flaw of predicate refinement is not the only reason of false positive results. While the proposed approach can successfully prove some specific paths to be spurious, it is not enough to fully prove correctness of some programs. However, the approach has further potential for improvements.

Keywords: static verification; predicate abstraction; thread-modular analysis.

For citation: Rudenchik V.P., Andrianov P.S. Predicate Abstraction Refinement in Thread-Modular Analysis. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 187-204. DOI: 10.15514/ISPRAS-2023-35(3)-14.

Уточнение предикатной абстракции при раздельном анализе потоков

В.П. Руденчик, ORCID: 0009-0000-6719-2594 <rudenchik@ispras.ru>
П.С. Андрианов, ORCID: 0000-0002-6855-7919 <andrianov@ispras.ru>

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

Аннотация. Комбинация анализа с раздельным рассмотрением потоков (Thread-Modular analysis) и предикатной абстракции является эффективной техникой верификации реального программного обеспечения. Одним из недостатков этой техники является уточнение предикатной абстракции при анализе многопоточных программ. В классической процедуре уточнения абстракции рассматривается только путь в одном потоке, и окружение Thread-Modular анализа не уточняется. Например, при применении эффекта из второго потока к первому путь к эффекту во втором потоке не уточняется.

Целью нашей работы была разработка более точной процедуры уточнения абстракции, которая бы переиспользовала имеющуюся процедуру уточнения абстракции и позволяла бы уточнять и путь в анализируемом потоке, и путь в окружении. Основная идея заключается в построении совместной логической формулы для двух путей. Так как имена переменных разных потоков могут совпасть, необходимо корректно переименовать и приравнять некоторые переменные для того, чтобы формула правильно отражала связи между потоками. Это позволяет получить предикаты, необходимые для доказательства недостижимости пути.

Предложенный подход был реализован на базе инструмента статической верификации CPAchecker. Подход был оценен на стандартном наборе задач SV-COMP и показал небольшое улучшение. Для больших программ улучшений в результатах не наблюдалась, так как описанный недостаток анализа не является единственной причиной ложноположительных результатов. Предложенный подход может успешно доказать недостижимость некоторых путей, то этого может быть недостаточно для доказательства корректности программы. Однако подход обладает дальнейшим потенциалом для совершенствования.

Ключевые слова: статическая верификация; предикатная абстракция; анализ с раздельным рассмотрением потоков.

Для цитирования: Руденчик В.П., Андрианов П.С. Уточнение предикатной абстракции при раздельном анализе потоков. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 187–204 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–14.

1. Introduction

Program verification is a process of checking if a program satisfies certain requirements. In static verification a program or its model is analyzed without actually running the code. There are multiple tools for program verification that implement various techniques targeted at different types of tasks. One of them is a reachability problem – a task of determining if a given point in a program is reachable. For reachability problem verification process can be broken down into two separate parts: 1) building a set of reached states; 2) checking if target state is in this set. While the second part is relatively simple, the first part is complex and resource-intensive. Various techniques and optimizations are developed to solve it. One of such approaches is abstraction.

There are many different types of analyses, which implement different kinds of abstractions. Using several abstractions at once can make analysis more efficient, especially for complicated pieces of code. CPA (Configurable Program Analysis) [1-2] was introduced as an approach of unifying different techniques for software verification (including abstractions). It allows combining different kinds of abstractions in various ways, so they can be used simultaneously and construct a more accurate model of a program.

In software verification approaches a model of a program is automatically extracted from the source code. It may not be accurate enough to prove certain properties of a program. Constructing more complex models is not always resource-efficient. This problem can be solved by using algorithms of iterative model refinement such as CEGAR [3], which refines abstractions using a counterexample. The algorithm iteratively refines the abstraction until it achieves a level of precision suitable for proving a specific property. Further, we will consider predicate abstraction [4], which assigns to each state a predicate that limits possible values of variables in the state.

Multithreaded programs traditionally cause additional problems for software verification. Classic approaches, which consider different combinations of thread interleavings, quickly result in state space explosion. There are other approaches, for example, Thread-Modular approach [5]–[7], which considers each thread separately in combination with some environment. The environment is constructed automatically during the verification process, and may be unique for every process. Thread-Modular approach demonstrates good performance and precision for industrial software as a target code. However, as we use abstraction technique, we need to have a refinement procedure. This presents a challenge, as the threads may interact with each other, for example, they may operate with the same shared variables or use local variables with the same names.

The paper presents a way of refining predicate abstraction in Thread-Modular approach. In Thread-Modular an error path is a path in a one thread, as threads are analyzed separately. However, it may contain effects from other threads, and there are paths to the effects in other threads. We introduce an efficient way to construct a joined boolean formula for two different thread paths. The idea is to rename local variables to avoid matching and add specific equalities of shared variables to represent dependencies of values of shared variables in different threads. Constructing a joined formula allows reusing a basic predicate refinement procedure to refine multiple paths all together. However, practical implementation poses some technical problems such as hanging caused by repeated analysis of the same path.

A current limitation of the approach is complicated thread interleavings. For example, if the analyzed thread interleaves with the second one that is also affected by the third one, the proposed approach might not be effective.

Experiments show that the approach allows refining more paths than the default predicate refinement procedure. It can successfully prove absence of errors for a certain number of tasks. However, the benefit is shown mostly on small artificial tests, as large real-world examples have a complicated thread interaction. Thus, even if the proposed predicate refinement procedure is able to remove some infeasible paths from abstractions, there are still other spurious paths due to other reasons, which do not allow to prove the correctness.

The main contributions are:

- an approach for environment refinement in predicate abstraction;
- implementation of the approach on top of the CPAchecker framework[1]. The source code is already merged in the main branch.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the theory. Section 3 contains a motivation example with a description of the problem. The proposed solution is presented in section 4. In section 5 some implementation features are described. Evaluation details are given in section 6, and section 7 contains brief information about related work.

2. Preliminaries

2.1 Software model checking

We consider a multithreading program as target software. This is a program, which contains more than one execution thread. The threads can operate with *local* variables, which are available only to specific threads, and *shared* variables, which are available to all threads. We do not specify any interface, like, POSIX, ARINC, or other, as it is irrelevant to our analysis.

Further, we consider software model checking approach for static verification. Such approaches allow the automatic extraction of a formal model from the source code and check it against predefined specifications or *properties*.

One of such properties is *reachability*. If a specific *error* state is reachable, then the property is violated and the program is incorrect. Accordingly, if no error state is found, the program is considered to be correct.

Another possible property is absence of *data races* [8]. Theoretically, it can be expressed via reachability [9], however in practice it is more efficient to consider it separately. Further, we will consider only reachability problem, as it is simpler. However, it is possible to apply the proposed refinement procedure for verification of other properties. Also, we do not consider any specifics of weak memory models [10].

[1] https://gitlab.com/p.andrianov/cpachecker/

2.2 Abstractions

As mentioned above, instead of analyzing a program itself we analyze a model of a program. Traditionally, a model of a program is a graph built upon Control Flow Automaton (CFA). The edges represent program operators from CFA and the states represent program memory, including location from CFA (pc) and assignment of values to all variables. The states are called *concrete* ones.

Even for a one integer counter possible values are numerous. Real-world software contains thousands of variables, and using *concrete* states in analysis leads to combinatorial explosion of a state-space. One of the ideas to reduce the number of considered states is *abstraction*. Abstract states represent multiple concrete data states. There are many different kinds of possible abstractions. Our approach is based on predicate one, so, further we will consider it. In predicate abstraction [3] an abstract state contains predicates over program variables. For example, abstract state ($x = 0$) represents many concrete states, including ($x \to 0$, $y \to 0$), ($x \to 0$, $y \to 1$), ($x \to 0$, $y \to 2$), etc. It constrains x to have a value of zero, but does not specify values of other variables. The same way abstract state ($x \geq 0$) $\wedge$ ($y \leq 1$) constraints variables x and y in the way defined by the predicates.

An operator *transfer* allows to build a next abstract state for a parent state and program operation (control flow edge). In predicate abstraction the operator *transfer* is the strongest postcondition of the parent state and program operation. A set of states, which are reachable by a *transfer* from some initial state, is a *reached set*. Note, that the *reached set* is a set of abstract states, and potentially, some abstract states may represent those concrete states, which are impossible in a real execution of a program. This is, because an abstraction is an *overapproximation* of a program. This is necessary for the soundness of an analysis i.e. in order for the program to not be falsely considered correct. *Reached set* is usually represented by Abstract Reachability Graph (ARG).

Abstraction is built with a certain *precision*: high *precision* means more precise abstraction. *Precision* is formally defined by an analysis. In predicate analysis a *precision* π is a set of predicates, which are used in constructing predicate abstract states. The lowest (the weakest) predicate precision is an empty set $\pi = \emptyset$. Predicate abstraction with the empty precision will contain only trivial predicate states $\top$, which corresponds to formula **True**. They represent any concrete state.

And how can the precision be changed? For example, if the abstraction is not precise enough and contains spurious paths, there is a need to refine it. This question will be addressed in the following section.

2.3 Refining predicate abstractions with CEGAR

As we have already described, abstraction is an overapproximation of a program, so, it may omit some details. Because of such imprecision, a program can be falsely considered incorrect. Therefore, there should be a way to refine the abstraction.

Counter-Example Guided Abstraction Refinement (CEGAR) [3] is an approach for increasing precision of an abstraction. It iteratively refines an abstraction using *counterexamples*. In case of reachability problem, counterexample is a path to an error state. Let us consider the way CEGAR refines the abstraction.

First, an initial abstraction (a set of reached states) is built with a given precision. By default the initial precision is set to the lowest precision, i.e to the empty one, meaning the abstraction is built imprecisely.

Then we should check if an error state is present in the abstraction. For the initial abstraction it means just syntactical reachability, as there are no valuable predicates. If the error state is unreachable, the program is correct and the analysis finishes.

If the error state is present in the abstraction, it does not mean that it is reachable in the program since the abstraction can be imprecise. The counterexample (a path to this state) needs to be checked

for feasibility precisely. If the error path is feasible in a precise model, the program is incorrect and the analysis finishes. If the error path is infeasible in the precise model, abstraction needs to be recomputed with the new precision provided by CEGAR. That is a default CEGAR loop. There are two points of interest here: how the counterexample is checked for precise feasibility and how new precision is obtained. Further we will consider these issues in case of predicate abstractions.

In predicate abstraction a *path formula* is calculated in order to check the counterexample for feasibility. Path formula is a conjunction of predicates that correspond to path operators. For instance, if a path contains three consecutive operators: an assignment operator $a = 1$, a conditional operator *if* $(a \geq 0)$ and another assignment operator $b = 1$, the corresponding path formula is $a = 1 \wedge a \geq 0 \wedge b = 1$. There is no contradiction in the formula, so it is satisfiable. Formula $a = 1 \wedge a < 0 \wedge b = 2$ corresponds to operators $a = 1$, *if* $(a < 0)$ and $b = 2$. This formula is unsatisfiable.

Satisfiability of a path formula is equivalent to existence of such input data (initial values of variables) that the error state is reachable. The satisfiability of the formula is checked by a specific external tool – SAT solver. If the formula is satisfiable, then the error state is considered reachable and analysis ends. Feasibility of the path in the abstraction but not in the program means that the abstraction is not precise enough and needs to be refined.

The way precision is extracted from a spurious counterexample depends on the abstraction. Moreover, there are different ways to refine predicate or any other abstraction. We are using Craig interpolation [11] to extract predicates from an unsatisfiable path formula. There is an interpolation theorem, which claims that for any logical formulas φ, ψ such that $\varphi \wedge \psi \equiv \bot$ there exists logical formula ρ, called an *interpolant*, such that every non-logical symbol in ρ occurs both in φ and ψ, $\varphi \rightarrow \rho$ and $\psi \wedge \rho \equiv \bot$.

In practice, we use interpolating solvers such as MathSAT [12], Z3 [13], or CVC5 [14], to calculate the interpolants. Being a conjunction of predicates, an unsatisfiable path formula can be split in two parts, usually in multiple ways, to satisfy the precondition of the theorem. Solver extracts multiple interpolants from a path formula, those interpolants are then added to precision. Note that interpolants are not the only way to extract new precision.

It is important to mention one of the optimizations for efficient abstraction rebuild. It is called *lazy abstraction* [15]. The main idea is to rebuild not all abstraction after refinement, but to identify the changed parts and reconstruct only them. During the refinement procedure a *refinement root* is identified. This is the state, which is a common parent of all changed subtrees in the reached set. The subtree is removed after refinement, and the analysis continues from the *refinement root*.

One more optimization, which also should be mentioned, is Adjustable Block Encoding (ABE) [16]. Its main idea is to avoid reconstructing predicate formulas in every state. Instead, formulas are constructed for every block that is composed of multiple states. Because of it, interpolants are usually not set for every abstract state. We do not need to describe this optimization in detail since it is irrelevant to our work.

This concludes an overview of predicate abstraction refinement with CEGAR. So far, we have only considered a path in a single thread. It is not immediately obvious how this refinement procedure can be applied to an analysis of multithreaded programs where a path contains operators from different threads. In the following section we describe an approach to analysis of multithreaded programs that can be combined with CEGAR.

2.4 Thread-Modular Analysis

Thread-Modular analysis [5]–[7] is an approach for verification of multi-threaded programs. Unlike algorithms that rely on complete enumeration of possible thread interleavings, Thread-Modular analysis uses an abstraction of thread interactions. It analyzes each thread individually with consideration of an *environment*, which is a model (abstraction) of possible effects that threads can

have on each other [7]. The more accurate the environment is, the more precise analysis is going to be. And less accurate and more abstract models can be used for analyzing large programs for which brute-force approaches are not applicable.

Interactions of threads can be formally described in terms of *projections*. A *projection of an operation* is an effect that the operation can have on other threads or an overapproximation of such effect. A projection can also contain a condition under which its effect can be applied. For instance, assigning a value to a local variable does not affect other threads, so a projection of this operation is empty. Now let us consider an assignment $x = 0$ to a global variable x. Its projection may contain the same assignment $x = 0$. Alternatively, a projection may be more abstract and contain assignment $x = *$, meaning "the thread can change a value of variable x to anything". Therefore, environment can be defined as a set of projections of all operators in the program.

While analyzing each individual thread, Thread-Modular analysis builds projections of every operator of this thread. The projections are part of the environment for other threads. After the primary analysis of each thread, Thread-Modular analysis considers an effect of the environment. For that purpose it checks each projection from the environment and each state in other threads if they are *compatible*, i.e. if the projection can be applied to the state. In predicate abstraction two predicate abstract states are considered compatible if a conjunction of their predicates looks satisfactory. If a projection and a state are compatible, the effect of the projection is applied to the state which results in creation of a new state called *applied state*. Projections express an effect of other threads, and applied states contain the effect, which is applied to the particular state in the current thread.

Applied states and the states that are reachable from them by operator *transfer* are added to the *reached set*. The state to which the projection was applied is considered to be a parent state of the applied state. Because of this, new paths are created that represent how threads interact with each other. Note that the applied state may be the same as the parent state, meaning the effect does not change anything.

An illustration of the approach is given in *Fig. 1*. There is a part of ARG representing the first thread and a part of ARG representing the second thread. Assignment operator $x = 0$ that follows state *B* from the second thread can be projected.

If the new *projection* is compatible with the state A from the first thread, it can be applied to the state *A*. The new *applied* state corresponds to application of the effect $x = 0$ to the first thread. The analysis continues in the first thread from the new *applied* state.

As Thread-Modular approach considers threads separately, the error path is also a path in a separate thread. However, the path may contain different effects, representing the thread interaction. The next section shows the problem during refinement of paths in the Thread-Modular case.

3. Motivating example

Let us consider the program in the *Fig. 2*. It contains two threads *thread*1 and *thread*2, both can change values of global variables *a* and *b*. The first thread assigns the value of 1 to variables *a* and *b* with mutex protection. Then it releases the mutex and checks that the value of *b* has not changed.

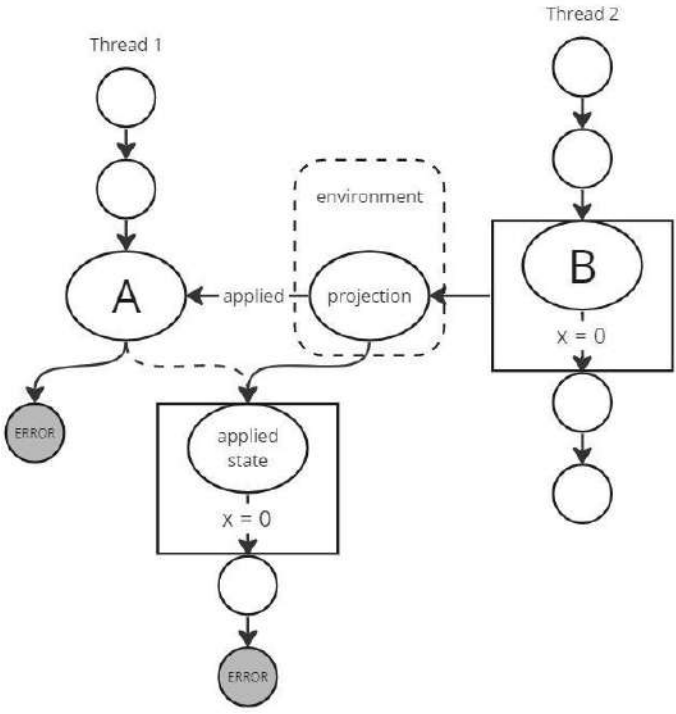


Fig. 1. Thread-Modular approach

```
 1 int a = 0, b = 0;
 2
 3 void *thread1(void *arg) {
 4     pthread_mutex_lock(&mutex);
 5     a = 1;
 6     b = 1;
 7     pthread_mutex_unlock(&mutex);
 8     assert (b == 1);
 9 }
10
11 void *thread2(void *arg) {
12     pthread_mutex_lock(&mutex);
13     if (a != 1){
14         b = 2;
15     }
16     pthread_mutex_unlock(&mutex);
17 }
18
19 int main(void) {
20     int t1, t2;
21     pthread_create(&t1, 0, thread1, 0);
22     pthread_create(&t2, 0, thread2, 0);
23     return 0;
24 }
```

Fig. 2. Example of a program

The second thread checks if the value of *a* has changed and if it has not, then it changes the value of *b* to 2; all while the mutex is locked. The error label (assertion in line 8) is not reachable, because

change of the variable b is allowed only in case of $a \neq 1$. However, analyzing the program with CPAchecker using Thread-Modular analysis with default predicate refinement returns a counterexample, meaning the error label is feasible. The reason this is happening is the inability to refine the predicate abstraction.

First, the analysis constructs a path to the error state. The path is in the first thread, as the error state (assert in line 8) is in the first thread. Initially, the predicate precision in empty, the path corresponds to operators $a = 1$ in line 5, $b = 1$ in line 6, and *assert* in line 8 and does not contain any effects.

The corresponding path formula: $(a = 1) \wedge (b = 1) \wedge (b \neq 1)$. It is unsatisfiable, because the value of b is not considered in the abstraction. So, the abstraction is successfully refined and the interpolant $b = 1$ is added to the predicate precision.

In the next iteration of the analysis another path is constructed. The path is in the first thread and it contains an application of the effect $b = 2$ (line 14) from the second thread right before line 8. The path corresponds to succession of operators $a = 1$ (line 5), $b = 1$ (line 6), $b = 2$ (line 14, thread 2), and *assert* in line 8. Actually, this effect cannot be applied since the operation $b = 2$ can only be executed if a does not equal 1 (line 13) but the value of variable a before line 8 is equal to 1.

The path is spurious, abstraction is not precise enough, because it does not contain any predicate over value of a. And the abstraction is supposed to be refined. But default refiner fails to prove that the effect cannot be applied.

The counterexample is shown in *Fig. 3* (highlighted in dark color). State A corresponds to the line 8, before assertion check. State B corresponds to the line 14 with operation $b = 2$. So, the projection represents the effect from operation $b = 2$ for other threads. It is indeed feasible as a path in a single thread if the projection is applied. But the projection could not have been applied. Default predicate refiner refines only a path to the error state, and it does not check the projection, state from which it was projected (state B in *Fig.3*) or a path to that state. Predicate abstraction of the second thread is not refined, and it stays not precise enough to exclude the application of the projection. Because of that, spurious counterexample is not ruled out.

If the predicate precision contained predicates $a == 1$ and $a \neq 1$ then state A would contain predicate $a == 1$ and both state B and the projection would contain predicate $a \neq 1$. That would make state A and the projection incompatible and the projection would have been applied. The question is, how to obtain such predicates.

4. Proposed solution

4.1 An approach overview

Let us consider a path to an error state in an abstraction. This is a path in a single thread, and it contains an applied effect, meaning it is affected by another thread. Let the path in the single thread be reachable in the abstraction. If the effect cannot been applied, the path is technically unreachable. One would naturally expect a refiner to detect the unreachability of the path and construct a more precise environment in which the effect would not be applied. However, the default refiner lacks the capability to do so as it only refines the path in a single thread and does not refine the environment. It is unable to prove that the effect cannot be applied. Therefore, the analysis considers the path feasible and the error label can be falsely recognized as reachable.

The problem arises, as the default refinement procedure considers only thread abstraction and misses the environment. So, we need an efficient way to refine two parts of the abstraction (thread and environment) together. And it means that the counterexample now consists of two parts: a path in thread (main path) and a path in an environment. If the two paths are spurious, we need to obtain interpolants that can potentially prove the incompatibility of these paths, and add them to precision. The next step is to determine an imprecise part of the abstraction and rebuild it with new precision.

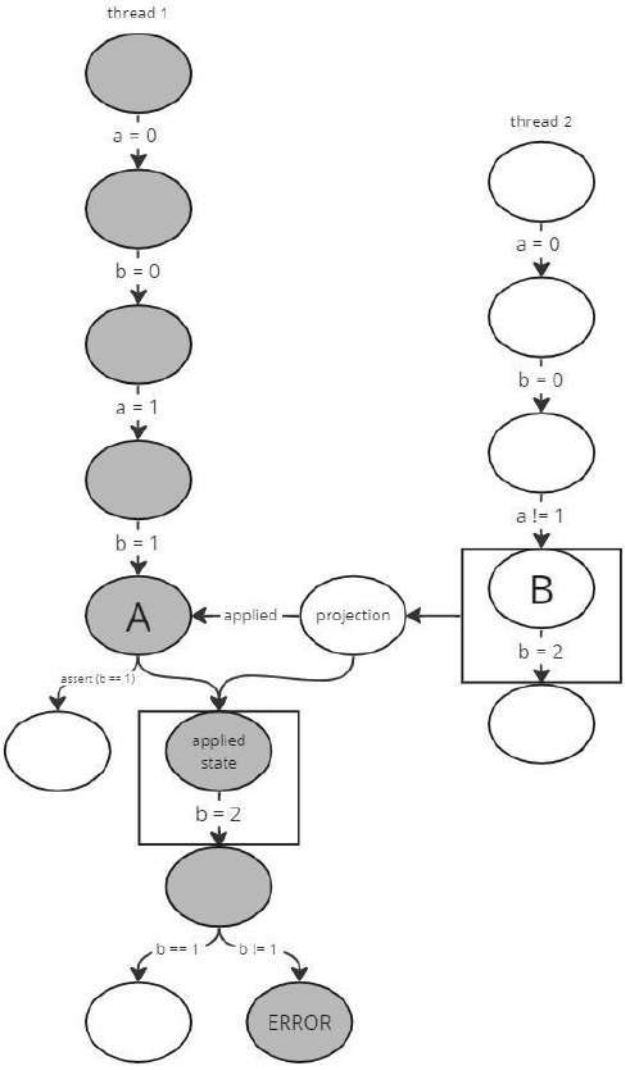


Fig. 3. Counterexample

One of the options to refine two paths is to develop a new refiner specifically for this task. However, this approach would lead to a considerable amount of code duplication, since only the refinement target is changed, not the refinement technique itself. Instead, we choose to extend an existing approach, and refine two paths altogether by reusing an existing refiner. While reusing a large piece of code is generally practical and efficient, it requires addressing certain issues to ensure successful code reuse. Since the input of a default refiner is a single path, the two paths need to be joined into one to be refined by it. Moreover, names of local variables may overlap, and global variables may appear in both paths, so they need to be carefully renamed in order to avoid false dependencies. Although the interpolation procedure stays the same, we still need some post-processing of obtained interpolants. Now we present the approach in more detail.

Consider an instance of a projection depicted in *Fig. 3*, where the projection originates from state B and is applied to state A. We consider two paths: the first is a path to state A and the second is the path to state B. The paths are reconstructed using ARG relations. A path formula, which is a conjunction of predicates that correspond to program operators, is constructed for each path, as it is

performed in default predicate analysis. In order to check simultaneous feasibility of these two paths, we check satisfiability of a conjunction of these two path formulas. Since the resulting formula is still a conjunction of predicates, it can be processed like a regular path formula of a single path. And then we request SAT solver about its satisfiability. However, the process is not as straightforward due to complications in joining the formulas.

4.2 Joining formulas

The path to the error state in *Fig. 3* contains multiple assignments to the same variable. For instance, $b = 0$ and $b = 1$ are successive assignments to variable b. If the path formula contained the unsatisfiable conjunction of the corresponding predicates $b = 0 \wedge b = 1$, it would be unsatisfiable regardless what other predicates it contains. Thus, path formulas are built with SSA indexation [17], which assigns an index to each variable that increments with each assignment. Variables with different indices are considered different. And since each variable is only assigned a value once, there are no collisions in path formula caused by multiple assignments.

SSA indexation can cause problems when joining formulas. Each thread (path) has its own SSA indexation. That means that a global variable can have multiple overlapping sets of indices, one for each thread. In a joined formula two instances of the same global variable from different threads but with equal indices will be considered as the same variable. This can cause unexpected dependencies. This problem can be solved by renaming variables in one of the threads.

For instance, we rename global variable *b* in the second (environment) formula to *env_b*. Adding a special symbol, which is not permitted in a variable name in real code, to the variable ensures that the newly renamed variable does not coincide with any other variable.

However, renaming loses relation between two threads, and we need to artificially restore it. Values of global variables at the point of projection application in both threads must be equal. In the opposite case, for example, if a global variable *b* in one thread is equal to 1 and in the second thread the same variable *b* is equal to 2, it means that the two states are incompatible. In order for a path formula to reflect that, we need to add variable equalities. Each global variable with the latest index in one thread is considered equal to this global variable with the latest index in the other thread. The equalities are then added to the joined path formula as new predicates in a conjunction. That ensures that the formula reflects relation between threads.

Another problem occurs if formulas contain local variables. There can be two local variables in different threads with identical names. When joined into one path formula they can potentially be treated as one global variable, which can affect satisfiability of the formula. To avoid that, all local variables of one of the two threads should be renamed. For instance, similarly to global variables, we rename local variable i in the second (environment) formula to *env_i*. However, we do not add any variable equalities for the local variables.

The resulting formula accurately represents two paths and a relation between them. If the formula is satisfiable then the two paths are considered feasible simultaneously and the error state is reachable. If this formula is unsatisfiable then the two paths are not feasible simultaneously and abstraction needs to be refined. The default Craig interpolation can be used to get interpolants. Usually, a path formula can be split into parts φ and ψ such that $\varphi \wedge \psi \equiv \bot$ in multiple ways. Interpolation is then performed for each partition to obtain more potentially useful predicates. The joined path formula is no exception. It is a conjunction of predicates and interpolants are extracted from it just like from any other path formula.

Let's take a closer look at predicates that are obtained during the interpolation. Let's consider a projection *proj* that was applied after state *A* and that was projected from the state *B*. Let μ_1 and μ_2 be path formulas for the paths to *A* and *B* respectively. If $\mu_1 \wedge \mu_2 \equiv \bot$ (meaning *proj* could not have been applied) then Craig interpolation theorem can be applied for such unsatisfiable conjunction. Therefore, there exists a predicate ρ_1 such that every non-logical symbol in ρ_1 occurs both in μ_1 and

μ_2, $\mu_1 \Rightarrow \rho_1$ and $\mu_2 \wedge \rho_1 \equiv \bot$. Since $\mu_2 \wedge \rho_1$ is an unsatisfiable conjunction, there exists predicate ρ_2 such that every non-logical symbol in ρ_2 occurs both in μ_2 and ρ_1, $\mu_2 \Rightarrow \rho_2$ and $\rho_1 \wedge \rho_2 \equiv \bot$.

Predicates ρ_1 and ρ_2 are then added to precision. A part of the abstraction is reconstructed with the updated precision (see lazy abstraction). In the default refinement procedure the rebuilt part of abstraction does not include states in the environment, but in order to eliminate the infeasible paths a part of the environment also has to be reconstructed. Predicate ρ_1 is an implication of path formula μ_1 which resembles a path to state A. Since predicate state is built as the strongest postcondition of the path, predicate state of state A will contain predicate ρ_1 in the rebuilt ARG. Likewise, predicate state of state B will contain predicate ρ_2. Since $\rho_1 \wedge \rho_2 \equiv \bot$, states A and B are now considered incompatible, and the projection cannot be applied. That proves infeasibility of the counterexample.

Finally, let's see how the counterexample in *Fig. 3* is refined. The first path is the path to state A and its path formula is $a_1 = 0 \wedge b_1 = 0 \wedge a_2 = 1 \wedge b_2 = 1$. Note, the subscript here is an SSA index. The second path is the path to state B and its path formula is $a_1 = 0 \wedge b_1 = 0 \wedge a_1 \neq 1$. By renaming variables in the second formula we obtain $env\_a_1 = 0 \wedge env\_b_1 = 0 \wedge env\_a_1 \neq 1$. After that we join the two formulas and add variable equalities: $a_2 = env\_a_1 \wedge b_2 = env\_b_1$. The resulting formula is

$$a_1 = 0 \wedge b_1 = 0 \wedge a_2 = 1 \wedge b_2 = 1 \wedge$$
$$\wedge env\_a_1 = 0 \wedge env\_b_1 = 0 \wedge env\_a_1 \neq 1 \wedge$$
$$\wedge a_2 = env\_a_1 \wedge b_2 = env\_b_1$$

Precise extracted interpolants depend on the solver and block encoding (see ABE). In theory, we can obtain interpolants $a_2 = 1$ and $env\_a_2 \neq 1$. The variables in the interpolants are then reverted to their original names, in our case by removing the prefix. Resulting predicates $a = 1$ and $a \neq 1$ are added to precision. In the rebuilt abstraction predicate state of state A would contain predicate $a = 1$ and predicate state of state B would contain predicate $a \neq 1$. Since $a = 1 \wedge a \neq 1 \equiv \bot$, states A and B are now incompatible, meaning the projection cannot be applied. That proves infeasibility of the counterexample.

4.3 Limitations of the approach

In theory new interpolants must exclude a spurious error path from the abstraction. Actually, an error path may be found again due to different reasons: optimizations, errors, unsupported cases, and so on. To avoid infinite loops of CEGAR loop, there is a technique for detection of repeated counterexamples. The default predicate refinement procedure compares error paths from last two CEGAR iterations and if they are equal stops the analysis. However, there are some difficulties in thread-modular case.

First, paths with effects can be falsely deemed equal. The default technique for detection of repeated counterexamples considers paths equal if their edges are identical, i.e. if paths correspond to the same sequence of executed operators. This approach does not take into account paths to effects if there are effects applied. For instance, two similar paths, each with different effects applied to the same state, are considered equal. The issue leads to false errors. In our approach this issue is more crucial since the environment can be refined and a new path can differ from the previous one solely based on paths in the environment.

Secondly, reusing the refiner multiple times in a single CEGAR iteration can lead to losing information about repeated paths, potentially resulting in looping. The default refiner procedure is run multiple times for one counterexample with applied effects. Both the path to the error state itself and the pairs of main paths and paths in the environment are refined, all within the same iteration. That interferes with error path detection. Default refiner only caches one path from the previous

refinement, and deletes it after comparing it with a next path. So, if a repeated counterexample contains an effect, the refinement procedure will be executed at least twice for it. The counterexample will be cached during the first execution but will be overwritten in the second one. As a result, the repetition of such a counterexample will go undetected, causing looping.

Caching all paths, which is an existing option, will not resolve the issue either. The same effect can be applied to the same state in different iterations. That means that the same joined path may be refined multiple times. However, that does not indicate repetition of counterexample and should not stop the analysis.

So far we have only considered a case where an error path contains only one applied projection that originates from a single state from the other thread. But in reality there might be several projections applied. If multiple projections are applied to the main path, meaning there are several effects applied to the first thread, we may iteratively check all of them one by one. If a main path and any path to one of these effects are not feasible together, the path is considered spurious and abstraction needs to be refined.

One more problem occurs when a projection is projected from multiple effects. Such projection can be created by the optimization which merges projections from different states into a single one. In that case all pairs of a path to each of these states and a path to the applied state are refined. In theory, the path should be considered spurious if at least one of the pairs of paths is infeasible simultaneously. But in reality, that projection merging optimization is not consistent with this theory. Because of this, we consider a projection application spurious if each path to each effect it was projected from is spurious.

Another problem occurs when projections are applied to the different threads. For example, one projection is applied to the first path, and a path to that projection in the second thread contains an effect from the third thread. The part of the environment that is important for the path to the projection will not be refined. The natural idea is to include recursion in the refinement process, but it is not yet clear if it would work somewhat effectively or work at all, considering other already existing limitations. The problem occurs when effects are applied not successively, multiple times and etc. Currently, this is a limitation of our approach.

5. Implementation features

The proposed approach was implemented on top of the CPAchecker framework as a separate predicate refiner. Its input is an error path in a main thread. First, the default refinement procedure is applied. If the path is spurious, the abstraction is refined with default predicate abstraction refinement procedure. It means that the contradiction is found in the path in one thread without any thread interaction. If the main path is feasible, it is analyzed with the proposed approach. For that purpose, we find all applied states in the path. An applied state is applied from a projection that can be projected from multiple states in another thread. For each such state the refiner checks feasibility of two paths: a path to the state in another thread and the main path.

It is important to note that the implemented approach differs from the presented theory. Theoretically, the first set of predicates should be obtained by interpolating a combination of two path formulas. That part is fully implemented in the actual code. However, the second set of predicates, in theory, should be obtained from interpolating a combination of path formula and the first set of predicates. Implementing this within the framework of the given task would be problematic. Given our decision to reuse an existing refiner which only input is a path, not a set of predicates; it would be quite a challenge to acquire these exact predicates. Nonetheless, the implemented method still has potential to prove infeasibility of a path.

One of the implementation features is refining two different combinations of paths. A main path and a path to an effect are concatenated in both possible ways and both combinations are refined. Solver extracts different predicates from these two constructed paths and both of these sets of predicates are necessary to prove spuriousness of the counterexample. Additionally, if two combinations of

paths are refined, the already existing code provides correct refinement root (a root of the subtree in the ARG that is rebuilt with new precision).

As it has been established, repeated counterexample detection is a problem. The same error path can be rediscovered again and again, which leads to hanging. To solve it, we integrated detection of repeated counterexample into our refiner. It checks if the last two paths in main thread are equal and caches the main path to the error state until next iteration. That effectively prevents looping.

The previously mentioned issue of paths being falsely regarded as equal also requires a suitable solution. In default repeated counterexample detection paths are considered equal if the (ordered) sets of executed operators are equal. Comparing paths by states is problematic since it would require caching a considerable part of ARG. We implemented an enhanced method of comparing paths by edges. Apart from edges in main paths it also compares edges in all paths to applied effects. It allows differentiating between paths with effects more effectively, but does not completely eliminate the possibility of false repeated counterexample detection.

6. Evaluation

The proposed approach was evaluated on standard benchmark set SV-COMP[2]. The benchmark set contains 161 tasks from directories:

- *pthread/;*
- *pthread-C-DAC/;*
- *pthread-divine/;*
- *pthread-ext/;*
- *pthread-memsafety/;*
- *pthread-atomic/;*
- *pthread-complex/;*
- *pthread-driver-races/;*
- *pthread-lit/;*
- *pthread-nondet/.*

The tasks are mostly artificially created tests with about 1 KLoc and 2-3 worker threads. They may contain a specific synchronization, like atomics, Dekker algorithms and others. We evaluated the new approach against two existing ones.

- **Default**. The default predicate refiner, which refines only one error path without considering other threads.
- **Simple**. The simplified version of refinement that checks feasibility of every path (including paths to effects) separately. Thus, it is more precise than **Default**, as it is possible to exclude paths to infeasible effects.
- **Effect**. The proposed approach for simultaneous refinement of two paths.

The tool was run with the thread modular approach over predicate analysis. The following options were used:

- precise encoding of environment actions;
- SMTInterpol is used for SAT check and interpolation;
- support for the same threads in tests.

The experiments were performed on a machine with Intel® Core™ i5-8250U CPU @ 1.60GHz × 8 and 8 GB of RAM, using 4 CPU cores; with Ubuntu 22.04.2 LTS. Timeout was set to 5 minutes. The results are presented in *Table 1*.

Table 1. Evaluation on SV-COMP benchmarks

Approach	**Default**	**Simple**	**Effect**
Correct results:	50	38	44
• Correct true	20	20	22
• Correct false	30	18	22
Incorrect results	71	51	52
• Incorrect true	0	0	0
• Incorrect false	71	51	52
Unknowns	40	72	65
• Timeouts	12	6	12
• Repeated Counterexample error	0	41	40
• Other Unknowns	28	25	13
CPUtime, s	6040	3969	6780

The proposed approach was able to prove correctness of two tests, which both thread-modular analysis and the simplified version of presented approach falsely considered incorrect. The simplified version didn't show any improved results.

The most frequently encountered error (both for **Effect** and **Simple**) was the repeated counterexample error, which indicates that the analyses recognized a counterexample as spurious but failed to refine the abstraction, leading to the counterexample being rediscovered. One possible explanation for this is that the obtained interpolants were insufficient to eliminate the path. Some errors were falsely reported due to the imperfect nature of repeated counterexample detection. At least three tests falsely reported a repeated counterexample error. The decreased amount of correct (and incorrect) false results is also caused by the repeated counterexample error.

As expected, the proposed approach is more time-consuming. Most of the extra time is spent on refining joined paths. The simplified version (**Simple**) averaged in less time than the default approach only because it reported repeated counterexample error almost immediately on several time-consuming tests.

The proposed approach was able to prove correctness of a motivation example (program in *Fig. 2*).

We also evaluated the approach on a benchmark set of more complicated tasks, based on Linux device drivers. Each task contains about 10 KLoc and about 5 threads. There are 7 such tasks. The proposed approach did not show any improvement, it mostly reported repeated counterexample error.

The reason for that is complicated thread interleavings. The proposed approach can refine specific paths, but eventually a path will be constructed that it cannot refine. A common example of such path is one where effects are applied to the different threads: one effect is applied from the second thread to the first thread and a path to that effect contains another effect application. However, in smaller tests, the predicates obtained during the first few iterations are typically enough to prevent such path from being constructed in the first place.

The evaluation results show benefit on a small subset of the benchmarks. The proposed approach did not show any improvement on complicated tests, since it is not targeted to analyze intricate

thread interleavings. While it can successfully prove infeasibility of counterexamples, this is often, but not always, not enough to prove correctness of a program. It works in a reasonable time and has potential for future improvement, as the issue with repeated counterexamples is mostly technical.

7. Related work

There are different approaches to the analysis of multi-threaded programs. They have different features and performance.

Precise approaches, based on bounded model checking techniques, investigate different techniques to reduce state space. The examples of the optimizations are partial-order reduction [18], context bounding [19-20], etc. They consider thread interleavings, and they do not have such problems with environment refinement. We do not dive deep into BMC approaches, and concentrate on thread-modular ones.

Thread-modular approach was first suggested by [21] and a predicate abstraction was composed with a thread-modular approach in [22]. There was only one thread in several copies, so, the environment of the thread is formed by itself.

An extension of the thread-modular approach, which also uses an abstraction, is firstly presented in [23] and then implemented in TAR [5]. One of the main difference is underapproximation of the environment. So, the approach does not need environment refinement.

A similar approach was also implemented in Threader tool [24]. Threader uses over-approximation for an environment, based on Horn clauses.

A thread modular approach to formal verification was presented in [25]. The idea is to provide invariants for every process, which together imply the formal requirement.

8. Conclusion

The paper presents an approach for predicate refinement in case of Thread-Modular analysis. The basic idea is to join thread-parted formulas into a single one, and check its satisfiability to determine whether two paths are feasible simultaneously.

Refinement of two paths in combination provides higher precision for the analysis. Because of this, specific spurious paths can be eliminated and a program can be proven to be correct. The evaluation results show benefit on medium-sized programs. Large programs contain intricate thread interleavings and the proposed approach is not enough to prove their correctness.

While the results show potential of the approach, there is room for future improvement. Some ideas for future work include recursive application of the approach to paths in the environment and improving the detection of repeated counterexamples.

Overall, the approach presented in this paper can be used in analyzing small and medium-sized multithreaded programs. It can successfully prove the correctness of programs that it is targeted at. Its efficiency can be increased by resolving technical problems that arise in its implementation.

References

[1]. D. Beyer, T. A. Henzinger, and G. The´oduloz, “Configurable software verification: concretizing the convergence of model checking and program analysis,” in Proceedings of CAV, (Berlin, Heidelberg), pp. 504– 518, Springer-Verlag, 2007.

[2]. D. Beyer, T. Henzinger, and G. Theoduloz, “Program analysis with dynamic precision adjustment,” in Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on, pp. 29– 38, sept. 2008.

[3]. M. Mandrykin, V. Mutilin, and A. Khoroshilov, “Vvedenie v metod CEGAR – utochnenie abstraktsii po kontrprimeram [Introduction to CEGAR – Counter-Example Guided Abstraction Refinement],” Trudy ISP RAN [Proceedings of ISP RAS], vol. 24, pp. 219–292, 2013.

[4]. S. Graf and H. Saidi, “Construction of abstract state graphs with PVS,” in Computer Aided Verification (O. Grumberg, ed.), (Berlin, Heidelberg), pp. 72–83, Springer Berlin Heidelberg, 1997.

[5]. T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, Thread-Modular Abstraction Refinement, pp. 262–274. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003.
[6]. A. Gupta, C. Popeea, and A. Rybalchenko, “Threader: A constraint-based verifier for multi-threaded programs,” in Proceedings of the 23rd International Conference on Computer Aided Verification, CAV’11, (Berlin, Heidelberg), pp. 412–417, Springer-Verlag, 2011.
[7]. P. Andrianov, “Analysis of correct synchronization of operating system components,” vol. 46(8), p. 712–730, Programming and Computer Software, 2020.
[8]. P. Andrianov and V. Mutilin, “Scalable thread-modular approach for data race detection,” Frontiers in Software Engineering Education, pp. 371– 385, 2020.
[9]. D. Kroening and M. Tautschnig, “Cbmc – c bounded model checker,” vol. 8413, pp. 389–391, 04 2014.
[10]. J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, “Software verification for weak memory via program transformation,” ESOP’13, (Berlin, Heidelberg), p. 512–532, Springer-Verlag, 2013.
[11]. W. Craig, “Three uses of the herbrand-gentzen theorem in relating model theory and proof theory,” Journal of Symbolic Logic, vol. 22, pp. 269– 285, Sep 1957.
[12]. R. Bruttomesso, A. Cimatti, A. Franze´n, A. Griggio, and R. Sebastiani, “The mathsat 4smt solver,” in CAV, pp. 299–303, 2008.
[13]. L. de Moura and N. Bjørner, “Z3: an efficient smt solver,” vol. 4963, pp. 337–340, 04 2008.
[14]. H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, Mohamed, M. Mohamed, A. Niemetz, A. No¨tzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, cvc5: A Versatile and Industrial-Strength SMT Solver, pp. 415–442. 01 2022.
[15]. T. A. Henzinger, R. Jhala, and R. Majumdar, “Lazy abstraction,” in Symposium on Principles of Programming Languages, pp. 58–70, ACM Press, 2002.
[16]. D. Beyer, M. E. Keremoglu, and P. Wendler, “Predicate abstraction with adjustable-block encoding,” in Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010, Lugano, October 20-23), pp. 189–197, FMCAD, 2010.
[17]. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph,” ACM Trans. Program. Lang. Syst., vol. 13, pp. 451–490, 10 1991.
[18]. P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Optimal dynamic partial order reduction,” SIGPLAN Not., vol. 49, pp. 373–384, jan 2014.
[19]. S. Qadeer and J. Rehof, “Context-bounded model checking of concurrent software,” in Tools and Algorithms for the Construction and Analysis of Systems (N. Halbwachs and L. D. Zuck, eds.), (Berlin, Heidelberg), pp. 93–107, Springer Berlin Heidelberg, 2005.
[20]. L. Cordeiro, J. Morse, D. Nicole, and B. Fischer, “Context-bounded model checking with esbmc 1.17,” in Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’12, (Berlin, Heidelberg), pp. 534–537, Springer-Verlag, 2012.
[21]. C. Flanagan and S. Qadeer, “Thread-modular model checking,” in Proceedings of the 10th International Conference on Model Checking Software, SPIN’03, (Berlin, Heidelberg), pp. 213–224, Springer-Verlag, 2003.
[22]. T. A. Henzinger, R. Jhala, and R. Majumdar, “Race checking by context inference,” in Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI ’04, (New York, NY, USA), pp. 1–13, ACM, 2004.
[23]. A. Malkis, A. Podelski, and A. Rybalchenko, “Thread-modular verification is cartesian abstract interpretation,” in Theoretical Aspects of Computing – ICTAC 2006 (K. Barkaoui, A. Cavalcanti, and A. Cerone, eds.), (Berlin, Heidelberg), pp. 183–197, Springer Berlin Heidelberg, 2006.
[24]. A. Gupta, C. Popeea, and A. Rybalchenko, “Predicate abstraction and refinement for verifying multi-threaded programs,” SIGPLAN Not., vol. 46, pp. 331–344, Jan 2011.
[25]. A. Cohen and K. S. Namjoshi, “Local proofs for global safety properties,” Form. Methods Syst. Des., vol. 34, pp. 104–125, Apr. 2009.

Информация об авторах / Information about authors

Вероника Павловна РУДЕНЧИК – студентка 5 курса специалитета механико-математического факультета МГУ, лаборант ИСП РАН. Научные интересы: статическая верификация, анализ многопоточных программ.

Veronika Pavlovna RUDENCHIK – 5th year student of the Specialist's program of MSU, faculty of mechanics and mathematics; laboratory assistant at ISP RAS. Research interests: software model checking, analysis of multithreaded software.

Павел Сергеевич АНДРИАНОВ – научный сотрудник ИСП РАН, кандидат физико-математических наук. Научные интересы: статическая верификация, параллельные программы.

Pavel Sergeevich ANDRIANOV – researcher in ISP RAS, Ph.D. Research interests: software model checking, parallel programs.

DOI: 10.15514/ISPRAS-2023-35(3)-15

Debugger for Declarative DSL for Telecommunication

[1] *T.M. Skazhenik*, ORCID: 0009-0002-1959-2010 <*taras.skazhenik@yandex.ru*>
[2] *D.V. Koznov*, ORCID: 0000-0003-2632-3193 <*d.koznov@spbu.ru*>
[1] *ITMO University*,
Kronverksky Pr. 49, St. Petersburg, 197101, Russia
[2] *Saint-Petersburg StateUniversity* ,
7-9 Universitetskaya Embankment, St Petersburg, 199034, Russia

Аннотация. Development of telecommunication product lines is still a very labor-intensive task, involving a great amount of human resources and producing a large number of development artifacts — code, models, tests, etc. Declarative domain-specific languages (DSLs) may reasonably simplify this process by increasing the level of abstraction. We use the term “declarative” implying that such a DSL does not enable the development of a closed software application, but rather supports creation, generation and maintenance of various kind of software assets — product database, events and event handlers, target code data structures, etc. At the same time, such a DSL may have some executable semantic, but it could be very specific and have many environment-wise requirements. Thus, execution and debugging of such DSL specifications is a meaningful task, which has no common solution due to the unique executable semantic. Consequently, it is not possible to use debug facilities of known DSL environments, such as xtext, MPS, etc. for such a case. In the current paper, we present a debugger for DevM — a declarative DSL intended for support device management in software development in the context of a router product line by a large telecommunication company. We clarify executable semantic for DevM, making it possible to execute DevM specifications in an isolated environment, i.e. in simulation mode, without generation of target code. We use a graphic model-based notation to depict every step of execution. Finally, we implement and integrate the debugger in the DevM IDE, using Debug Adapter Protocol and language server architecture combined with the Eclipse xText/EMF tool chain.

Key words: product lines; telecommunication systems; DSLs; debugging; IDE.

For citation: Skazhenik T.M., Koznov D.V. Debugger for Declarative DSL for Telecommunication. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 3, 2023. pp. 205-214. DOI: 10.15514/ISPRAS-2023-35(3)-15.

Отладчик декларативного DSL для разработки телекоммуникационных систем

[1] *Т.М. Скаженик,* ORCID: 0009-0002-1959-2010 <*taras.skazhenik@yandex.ru*>
[2] *Д.В. Кознов,* ORCID: 0000-0003-2632-3193 <*d.koznov@spbu.ru*>
[1] *Национальный исследовательский университет ИТМО,*
Россия, 197101, Санкт-Петербург, Кронверкский пр., д. 49, лит. А.
[2] *Санкт-Петербургский государственный университет,*
Россия, 199034, Санкт-Петербург, Университетская наб., д. 7-9.

Аннотация. Телекоммуникационные системы являются одними из самых трудоёмких видов ПО, вовлекая большое количество людей, денежных средств, а также времени. Декларативные предметно-ориентированные языки (DSLs) могут существенно помочь в разработке таких систем, реализуя

подходящие абстракции. Мы используем термин «декларативные», подразумевая, что программы на таком DSL предназначаются не для программирования исполняемой логики, а для описания данных (базы данных сетевого устройства, структуры данных целевого кода и т.д.) и задания некоторого модельного поведения устройств при возникновении определённых событий. Таким образом, исполнение таких программ в целях отладки невозможно осуществить, сгенерировав и запустив конечный код, т.е. не удаётся использовать средства типа xtext, MPS. Между тем отладка таких спецификаций является востребованной задачей в виду объёмности спецификаций (десятки тысяч строк кода), а также большого числа точечных изменений, вносимых при copy/paste, в ходе разработки очередной телекоммуникационной системы, принадлежащей данному семейству продуктов.
В предлагаемой статье описывается отладчик для предметно-ориентированного декларативного языка DevM. Этот язык предназначается для описания базы данных аппаратуры роутеров и свичей, задания специфической информации, необходимой для инициализации драйверов устройств, и описания высокоуровневого поведения системы при получении специфических событий из сети и от аппаратуры самого устройства. Язык ориентирован на использование в контексте семейства телекоммуникационных систем одной крупной телекоммуникационной компании. В рамках работы над отладчиком была уточнена исполняемая семантика DevM для задания событийно-ориентированного поведения системы, а также введена специальная модель (т.е. создана новая часть языка DevM) для задания отладочной конфигурации отлаживаемой системы. Исполнение программы на DevM выполняется без генерации целевого кода. Для наглядного отображения шага исполнения программы использовалась графическая событийно-ориентированная нотация. Интеграция созданного отладчика с DevM-фреймворком была выполнена с помощью Debug Adapter Protocol и языкового сервера DevM (language server), разработанного с помощью стека технологий Eclipse xText/EMF.

Ключевые слова: семейство программных продуктов; телекоммуникационная система; предметно-ориентированный язык; отладка; среда разработки.

Для цитирования: Скаженик Т.М, Кознов Д.В. Отладчик для декларативного DSL для разработки телекоммуникационных систем. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 205–214 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–15.

1. Introduction

Nowadays, it is typical for large companies to develop not a single software product but a number of products with varying features and functionality, providing upgrades, etc. All of these products and corresponding development infrastructure form a product line [1]. This approach expands the market capacities of a company and provides reuse of various development assets, e.g. code, models, requirements, tests, etc. Following the trend, a large telecommunication company is developing a product line of network routers. The product line contains about fifty different products, hundreds of unique boards, several hundred thousand C files, and more than ten million lines of source code. One of the problems of a product line is the development of the Device Management layer. This layer focuses on hardware drivers and network interfaces of the router being provided to network management layer. The problem is in a large range of hardware, complicated hardware connections (in particular, it is possible to insert various cards into the motherboard of the router) and various configurations of one product depending on demands of customers. To meet these problems, a special declarative DSL was developed [2]. This language provides the ability to specify hardware structure of the product that is visible to software. Furthermore, it can also specify the behaviour of a product in an event-based manner. It provides abstractions to define various product information, supporting generation of product configuration, network data, events and event handlers, target code data structure, etc. A special IDE that fully supports the proposed DSL was developed. Finally, a debugger was needed to improve maintenance of DSL programs [3]. Leading DSL environments such as Xtext [4], GEMOC Studio [5], and MPS [6] support a two-level debug model [7] that is not suitable for declarative DSLs. Moreover, debug development facilities that are provided within these environments are deeply integrated with them, and their transfer to other runtime platforms is highly limited. Microsoft Visual Studio Code supports the Debug Adapter Protocol that provides a standard for the debugger user interface rather than technologies for development. Thus, DSL debugging for

declarative languages is a pressing problem. There is a number of research papers concerning DSL debugging [3, 8], but they do not deal with event-based behaviour DSLs. Event-based debugging is implemented in a series of model-based development toolsets such as YAKINDU [9], Rhapsody [10], but these tools are oriented at the UML-based system structure (components, interfaces, ports, channels, etc.). In the case of DevM DSL, we have both a specific system structure model. We may conclude that existing research and development tools do not provide any significant basis for developing a debugger for event-based declarative DSLs.

Thus, creating it is going to be research-intensive. The novel contributions of our paper are as follows:

- Scenario-oriented debugging concept for DevM dsl
- Use cases of the debugger
- An extension of DevM for configuration and initialization of system developed for debugging
- Graphical model-based notation for visualization of debug execution trace
- Implementation of the debugger with the support of Debug Adapter Protocol and integration into the DevM framework.

This article is organized as follows. Section 2 provides some background of the research. Section 3 presents scenario-oriented debugging concept for DevM and use cases of the debugger. Section 4 describes extension of DevM for specifying debug configuration of the product. Section 5 introduces graphical model-based notation for visualization of debug execution trace. Section 6 describes debugger implementation issues. Section 7 contains an overview of related work, and finally, section 8 provides the conclusions of the paper.

2. Background

The software part of the router in the considered product line consists of two main components: network management and device management. The latter encompasses hardware drivers and a network agent that provides an intermediate level between the drivers and the network management component. It implements a set of rules that determine the router's reaction to various network management events. The domain-specific language DevM is intended for describing the Device Management subsystem. DevM consists of the following parts:

- *Composition model* aims at describing hardware part of the router that is visible for drivers and network management. It consists of a set of boards and cards. The latter are a special type of boards and can be inserted into boards' or other cards' special slots, extending the functionality of the parent device. Actually, DevM specification of the product describes a set of board and card types (moduleTypes). A real configuration of the product delivery depends on customer requirements — that is, similar to the variability of hardware units in a laptop, when the customer just specifies type of the storage, volume of RAM, etc. during their purchase. Thus, facilities for creating target product configurations are outside of the DevM due to including not only device management level information. Some features of DevM for creation of debug configurations (debug model) will be described later.
- *Inheritance Model* addresses to specifying network management attributes of hardware elements.
- *Behaviour Model* focuses on event-driven behaviour of the network agent.

Let us consider the behaviour model in more detail. Specification of the network agent behaviour consists of a set of rules. Each rule includes the event that the network agent is subscribed to. The event triggers the action sequence if the logical condition attached to the event is true. The following kinds of actions are allowed: create an alarm event, log information, restart the network agent,

change attributes of the hardware elements of the router, as well as, possibly, other elements on the network.

It should be noted, DevM was designed to describe router hardware structure and special data structures including various configuration information. DevM does not actually let the user specify software's control flow, whereas DevM specification is not a closed executable specification although it includes some behaviour facilities. Moreover, various parts of DevM specification generate various assets, including data for the router database, C data structures and function signatures, etc. But generated C code is not closed and ready to be executed. A significant part of device management code is implemented manually.

Thus, it can be said the DevM is a declarative domain specific language. It should be stressed we do not imply logical programming facilities, but take into account to the fact that system code generated on DSL is not closed and consequently executed. A lot of other code is needed to execute it, and this additional code is developed outside the suggested DSL.

Nevertheless, declarative DSL could contain some part, which have executable semantic and may be launched in some simulation environment. This simulation (debugging is a special case of such simulation) may have a sense for DSL users helping to clarify dark corners of the DSL specification or finding errors.

The complete grammar of DevM is an Extended Backus-Naur Form (EBNF), which was created via XText [4]. Based on this grammar, an IDE language server is generated. DevM language server is integrated to Visual Studio Code, where an IDE interface is implemented. Visual Studio Code as a target environment is an external requirement to DevM.

3. Debug Concept and Debbuger Use Cases

In our case, we need a way to execute an event-based specification for a single component — that is, the device management agent. The behaviour of this agent is set using the behaviour model defined for the product with DevM tools. The device management agent receives events from outside — as in, from the network, as well as from the hardware of its router. In addition, the agent can create events for itself and process them itself too.

Being dependent on the environment, the device management agent must correctly process events received from it. It is this aspect that is interesting from the point of view of the debugger, since the processing of one external event is a purely internal matter of the device management agent, and it does not require any additional data from outside. Thus, emulation of receiving such an event could be the start of a debug section run by the developer in order to test the agent's handling of it. It is important to understand that the agent can be in different states, in each of which it must correctly process such an event. For example, it can receive a request from the network for reconfiguration and router restart either in a normal, regular state, or in a state of reduced bandwidth. Accordingly, two different rules are required to process the same event, and they correspond to different specifications of the initial state of the agent and different debug sessions.

During the processing of a single external event, the device management agent can activate more than one rule. This happens via the mechanism of the agent creating events for itself, searching for a suitable rule and executing it. Accordingly, the debug session ends when all rules are executed, and the device management agent message queue is empty.

Let us explain why the device management agent generates events for itself. It is due to the fact that the behaviour model is composite: different rules are created at different levels of the product's decomposition, for example, at the level of chips included in the board, or at the level of ports. Specifying chips and ports, it is important to determine how the processing of various events addressed to them takes place. At the same time, the exact origin of these events is not considered – be it the network or the top level of the device management agent. These rules can also be created by different developers responsible for managing different hardware units of the router. Moreover,

the same rule can participate in various scenarios, and in this case, rules are used for behaviour decomposition and reuse.

Note also that the behaviour model may differ for different configurations of the product, since they may include different types of equipment.

We have identified the following DevM debug use cases:

- Exploring the product configurations for a specific customer without a target platform, i.e. on a DevM developer workstation.
- Considering a subset of product configurations during DevM development to detect possible bugs. It is important to find bugs exactly on the development level they are made on. If these bugs are detected on the following development levels, the cost of bug detection will increase.
- Analyzing a specific product configuration in the situation when some bug occurs. It could be possible that the reason for the bug is contained in the DevM specification. If it is not so, the next development level should be explored.

4. Debug Model

In order to run a debugger on a behaviour model of the product, it is required that the user precisely defines the debug scenario: product configuration, current state, and debug event. This is done with the DevM language, which has been suitably extended for this purpose.

In order to define the hardware product configuration used in this debug scenario, the appropriate moduleTypes defined in the main DevM product specification are instantiated and the relationships between these instances are specified. The latter means that cards are inserted into appropriate slots of boards and possibly other cards. By this means, a tree of real devices of the product is built. All necessary attributes of each device from this tree are then set – DevM has also been extended for this purpose.

State of product configuration refers to setting values attributes, specifying the required current state of the product configuration.

A debug event specifies the start event that triggers the debug scenario.

Below is a simplified example of a debug scenario for the case of “restarting” the router when the voltage in the system drops” (see *Listing 1*). This scenario is described in the special debug_scenario1 package, which imports the core package of this product, containing the definitions of the main moduleTypes of the product.

The composition section describes the product configuration, which consists of the main_board1 and card1 inserted into the main_board1 in a slot called card_slot1. Note that the voltage sensor is installed on the card, as follows from the type description of this card in the main DevM specification of the product. Further, it is indicated that there is one external 100 Gbit port port1, into which the split4_25 optical converter is inserted, splitting this port into four 25 Gbit ports.

Further, in the attributes section, the state of the specified product configuration is set: main_board1, card1, sensorT have the “ready for operation” status, and card_slot1 is connected to power; sensorT also has a valid value of 12; the first of the 25 gigabit ports is activated (i.e. through it, the router communicates with the network).

Finally, in the event section, the event that triggers this debug scenario is set: the voltage measured by sensorT becomes invalid (of value 9, but interval allowed is from 12 to 15). The behaviour model has a rule which is activated when the voltage is below 12, see *Fig. 1*. It is triggered by the changing sensor’s attribute from 12 to 9. In the context of this rule an alarm “Low voltage” is exposed and another event is created. The last is done by changing the attribute card1.port1.port25GE.IS_AVAILABLE from 1 to 0, meaning the active port is disabled. The second rule create alarm “Port is down”.

```
package degug_scenario1 {
  import core;

  composition {
    BoardHardType main_board1;
    CardHardType card1;
    main_board1.card_slot1 <- card1;
    card1.port1 = split4_25;
  }
  attributes {
    main_board1 {
      IS_AVAILABLE = 1;
    }
    card1 {
      IS_AVAILABLE = 1;
    }
    main_board1.card_slot1 {
      POWER_STATUS = 1;
    }
    card1.sensorT {
      IS_AVAILABLE = 1;
      VLT_CURR_VALUE = 12;
    }
  }
  override attributes {
    card1.port1.port25GE {
      IS_AVAILABLE = 1;
    }
  }

  event {
    modify card1.sensorT
           : VLT_CURR_VALUE = 9;
  }
}
```

Listing 1. Debug scenario.

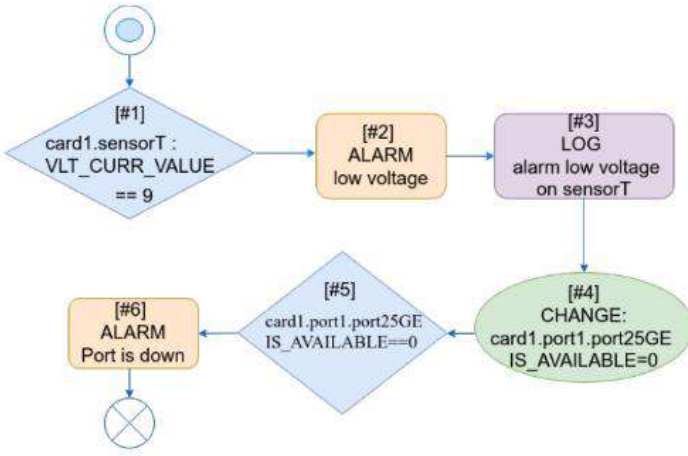


Fig. 1. An example of graphical model-based notation for visualization of debug execution trace

5. Visualization of debug results

Let us now consider the graphical model-based notation for visualization of the debug execution trace. As mentioned above, such a trace visualizes the step-by-step execution of the rules involved in the debug scenario. *Fig. 1* shows an example of such a diagram. It starts with a Start symbol (double circle filled in blue inside).

It is followed by the first event that triggered this scenario. Note that events in the DevM behaviour model are changes of the attributes of the device database on the router. The corresponding router devices are subscribed to changes of certain attributes; therefore, these devices have rules that start with this event. Device management agent combines all of these rules to whole behaviour model as described above. There can be multiple rules for handling the same event, but then they must differ in conditions that immediately follow the event. An event is denoted by a blue diamond.

Further, the brown rectangle denotes an alarm, the lilac one — logging, and the green oval indicates network device attribute changes. These changes, in turn, can cause further events to be fired for which a suitable rule is found. After the execution of the last rule, the end symbol of the debug scenario is drawn — a circle with crossed lines. At the top of each graphical symbol, except for the start and end, the step number is indicated. The user executes the debug scenario step by step, and as a result of each step, the corresponding graphic element is drawn in the diagram.

6. Debugger Implementation

The debugger implementation scheme is shown in *Fig. 2*. The debugger is divided into two parts: the Debugger Back End, which performs debugging and is integrated into the DevM language server, and the Debugger Front End, which implements the user interface and is integrated into the Visual Studio Code DevM plugin. These parts interact via the standard Debug Adapter Protocol, which passes debug commands from user to debug back end and debug information (attribute values. etc.) from back end to user the user to view.

The main difficulty was the implementation of the Debugger Backend. It consists of the following components: ConfigProcessor, DebugController, DebugSession, Variables Control System.

The *ConfigProcessor* component processes DevM specification of the debug scenario DevM specification or the whole product, transforming them into a convenient representation: namely, the device tree of a given product configuration based on hardware connections. This abstraction provides a structure that uniquely defines the “parent-child” relationship, which is important for searching in the behaviour model.

The *DebugController* component connects the Debugger Frontend and Debugger Backend, providing an API to initialize the debug session. When a request is received to start a debug scenario, the DebugController processes the incoming debug configuration using the ConfigProcessor, and creates an instance of the DebugSession based on the received data. Next, the controller redirects the request received from the front end to the DebugSession, and upon completion of the action sends the result back to the Debugger Frontend side.

The *DebugSession* component is the main debug engine. It implements various debugging steps, and also provides control over the storage and updating of data that is relevant for each step. Unlike general-purpose languages, where the program, as a rule, is executed on some hardware device, DebugSession simulates the entire execution process. Thus, it is easy to support the rollback of steps, which is a difficult task in the general case.

The *Variables Control System* component is a collection of classes responsible for storing, processing and transforming debugging information. The tasks of this component are the following: ensuring correct persistent storage of values and attributes of the router; splitting data into stack frames corresponding to the debug state at a certain step; serialization of objects into a representation that specifies the nodes of the debug graph. Thus, the component acts as a universal delegate for working with data stored during debugging.

7. Related work

The need for debugger development tools for DSLs is recognized by the community. Due to this, XText [4], GEMOC Studio [5], and MPS [6] as well as other DSL environments support meta-debug facilities. However, these facilities are oriented at executable DSLs, which have strict executable semantics and can be generated into Java and other industrial programming languages. Very often in this case, a two-level debug model is used [7]. It means that real debug is performed for generated DSL code, and special tools just raise debug information to the DSL level and accept the corresponding user commands from there. This approach is not suitable for our case of various program assets being generated according to the DSL specification, as they do not form a closed executable application.

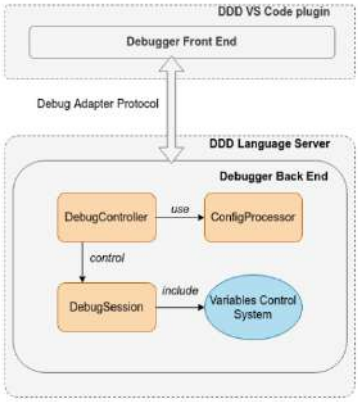


Fig. 2. Debugger implementation schema

There are studies on creating meta debug facilities for more complex cases by declaratively specifying executable semantics of the DSLs [3, 8]. However, these studies are at their pilot stages and cannot be employed in the industry. In addition, using this approach, it is difficult to express event-oriented executable semantics, which is important for our case.

Event-oriented debugging is implemented in a series of model-based development toolsets for real-time systems such as YAKINDU [9] and Rhapsody [10]. Such toolsets support UML statecharts and provide facilities for debug statecharts inside of the modeling environment. But, first, these solutions are deeply integrated into the toolsets and cannot be reused. Second, they are oriented at the UML-based system structure (components, interfaces, ports, channels, etc.). In practice, they provide execution and debug for a set of communicated components including statecharts. This execution model is redundant for our case, since we are executing a fragment of one component. In addition, we have a significantly different structure model.

Thus, we can conclude that creation of debuggers for declarative industrial DSLs is an open task that does not have a ready-made solution. Separate tools can be used for solving it, for example, the Debug Adapter Protocol and templates for creating the debugger front end. But the majority of work is in specifying the executable semantics for that part of the DSL that makes sense to debug, as well as support the corresponding executable environment in the DSL IDE.

8. Conclusions

In this paper, we have proposed a debugger for the DevM declarative language, which is intended for the development of device management components of a router product line of a large telecommunication company. As a continuation of this work, we plan to focus on increasing the number of actions used in the rules, as well as adding support for new features of the behaviour model that will be introduced in the future.

References

[1]. P. Clements, L. M. Northrop, Software product lines – practices and patterns, SEI series in software engineering, Addison-Wesley, 2002.
[2]. E. Semenov, S. Kai, C. Gen, D. V. Luciv, D. V. Koznov, Visual Language for Device Management in Telecommunication Product Line. MEDI Workshops 2021, pp. 204–216.
[3]. R. T. Lindeman, L. C. L. Kats, E. Visser, Declaratively defining domain-specific language debuggers, in: E. Denney, U. P. Schultz (Eds.), Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011, Portland, Oregon, USA, October 22-24, 2011, ACM, 2011, pp. 127–136.
[4]. Eclipse Project, XText, 2022. URL: https://www.eclipse.org/Xtext/.
[5]. GEMOC, 2022. URL: https://gemoc.org.
[6]. MPS: Meta Programming System, 2022. URL: https://www.jetbrains.com/mps/.
[7]. M. Kartashov, Two-level debugging, System Programming 1 (2005), pp. 348–365(In Russian).
[8]. A. Chis, M. Denker, T. Gîrba, O. Nierstrasz, Practical domain-specific debuggers using the moldable debugger framework, Comput. Lang. Syst. Struct. 44 (2015), pp. 89–113.
[9]. Itemis AG, YAKINDU, 2022. URL: https://github.com/Yakindu.
[10]. IBM, Rhapsody, 2022. URL: https://www.ibm.com/docs/en/rhapsod, (accessed: 01.05.2023).

Информация об авторах / Information about authors

Тарас Михайлович СКАЖЕНИК – студент второго курса магистратуры университета ИТМО. Сфера научных интересов: программная инженерия, телекоммуникационные системы, отладка, машинное обучение.

Taras Mikhailovich SKAZHENIK – second-year master-student of ITMO University. Research interests: software engineering, telecommunication systems, debugging, machine learning.

Дмитрий Владимирович КОЗНОВ – доктор технических наук, профессор кафедры системного программирования СПбГУ. Сфера научных интересов: программная инженерия, модельно-ориентированная разработка программного обеспечения, программные данные, машинное обучение.

Dmitry Vladimirovich KOZNOV – Doctor of Technical Sciences, Professor of the Software Engineering Chair, St. Petersburg State University. Research interests: software engineering, model-driven software development, program data, machine learning.

DOI: 10.15514/ISPRAS-2023-35(3)-16

Analyzing Hot Bugs in the Linux Kernel by Clustering Fixing Commit Messages

S.M. Staroletov, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
N.A. Starovoytov, ORCID: 0009-0007-0242-0198 <nikstarall@gmail.com>
N.A. Golovnev, ORCID: 0009-0008-0258-4560 <kolya.golovnev@mail.ru>

Polzunov Altai State Technical University
46, prospect Lenina, Barnaul, Altai region, 656038, Russia.

Abstract. In system software environments, a vast amount of information circulates, making it crucial to utilize this information in order to enhance the operation of such systems. One such system is the Linux kernel, which not only boasts a completely open-source nature, but also provides a comprehensive history through its git repository. Here, every logical code change is accompanied by a message written by the developer in natural language. Within this expansive repository, our focus lies on error correction messages from fixing commits, as analyzing their text can help identify the most common types of errors. Building upon our previous works, this paper proposes the utilization of data analysis methods for this purpose. To achieve our objective, we explore various techniques for processing repository messages and employing automated methods to pinpoint the prevalent bugs within them. By calculating distances between vectorizations of bug fixing messages and grouping them into clusters, we can effectively categorize and isolate the most frequently occurring errors. Our approach is applied to multiple prominent parts within the Linux kernel, allowing for comprehensive results and insights into what is going on with bugs in different subsystems. As a result, we show a summary of bug fixes in such parts of the Linux kernel as kernel, sched, mm, net, irq, x86 and arm64.

Keywords: bugs; Linux; clustering; fixing commits; kernel.

For citation: Staroletov S.M., Starovoytov N.A., Golovnev N.A. Analyzing hot bugs in the Linux kernel by clustering fixing commit messages. Trudy ISP RAN/Proc. ISP RAS, vol 35, issue. 3, 2023., pp. 215-242. DOI: 10.15514/ISPRAS-2023-35(3)-16

Анализ актуальных ошибок в ядре Linux путем кластеризации сообщений об исправлениях в git-репозитории

С.М. Старолетов, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
Н.А. Старовойтов, ORCID: 0009-0007-0242-0198 <nikstarall@gmail.com>
Н.А. Головнев, ORCID: 0009-0008-0258-4560 <kolya.golovnev@mail.ru>

АлтГТУ им. И.И. Ползунова
Россия, 656038, Алтайский край, г. Барнаул, пр. Ленина, 46.

Аннотация. В средах системного программного обеспечения циркулирует огромное количество информации, поэтому крайне важно использовать эту информацию для улучшения их работы. Одной из таких систем является ядро Linux, которое не только поставляется с полностью открытым исходным кодом, но и предоставляет исчерпывающую историю о разработке в своем git-репозитории. Здесь каждое логическое изменение кода сопровождается сообщением, написанным разработчиком на естественном языке. Обрабатывая данные репозитория, мы сосредотачиваемся на коммитах с сообщениями об исправлении ошибок, поскольку анализ их текста может помочь выявить наиболее распространенные типы ошибок. Основываясь на наших предыдущих работах, в этой статье мы предлагаем использовать методы анализа данных. Для достижения наших целей мы предлагаем

различные методы обработки сообщений в git-репозиториях и используем автоматизированные методы для выявления распространенных ошибок в них. Вычисляя расстояния между сообщениями об исправлении ошибок, превращая их в вектора и группируя в кластеры, мы далее можем эффективно классифицировать и выявлять наиболее часто возникающие ошибки. Наш подход применяется к нескольким важным частям ядра Linux, что позволяет понять, что происходит с ошибками в различных его подсистемах. В результате мы показываем сводку исправлений ошибок в таких частях ядра Linux, как kernel, sched, mm, net, irq, x86 и Arm64.

Ключевые слова: ошибки; Linux; кластеризация; исправляющие коммиты; ядро.

Для цитирования: Старолетов С.М., Старовойтов Н.А., Головнев Н.А. Анализ актуальных ошибок в ядре Linux путем кластеризации сообщений об исправлениях в git-репозитории. Труды ИСП РАН, том 35, вып. 3, 2023 г., стр. 215-242 (на английском языке). DOI: 10.15514/ISPRAS–2023–35(3)–16.

1. Introduction

In today's software development landscape, closed systems are no longer able to compete with open ones due to the involvement of highly skilled users who can not only test the software but also understand its code and suggest changes. The git version control system and its associated services are based on a fork and pull request approach, allowing users to easily propose changes and administrators to accept them after reviewing diffs. Git is designed for distributed work and encourages local changes, with each commit being accompanied by a comment about what was done. Originally created by Linus Torvalds for coordinating the development of the Linux kernel, git has become a super successful project.

While system program code development direction is not that popular among most modern software developers mainly due to the scarcity of qualified engineers, there is a large amount of data circulating in system software environments that can be analyzed using popular data analysis methods. This paper proposes to analyze commit messages in the development of the Linux kernel using automated methods. With a large number of commit messages available, common patterns can be automatically identified from the big data in natural language. The focus is on identifying and correcting the most typical errors in system software.

The Linux OS, based on an open modular kernel concept by Linus Torvalds, is constantly evolving with contributions from a large number of developers, both individuals and representatives of leading companies in the industry. All changes are made by committing them to developers' gits, and some eventually become available in the mainstream kernel at Torvalds GitHub. Such commits are usually verified by leading developers using the pull-request mechanism. Therefore, commit data analysis can provide insights into the evolution of the kernel.

The objective of this work is to automatically analyze commits in the Linux kernel repository to identify the most representative bugs. The paper discusses and explores data analysis methods for Linux commit messages.

The present paper is an extension of the report presented at SYRCoSE Software Engineering Colloquium 2023 in Penza [1].

2. Related work

In the pioneering work by Chou, Yang, Chelf and Engler [2] as well as ten years later by Palix, Thomas, Saha, Calves, Lawall and Muller [3], static analyzers were used to automatically check for potential errors in the Linux kernel code based on a given configuration over different kernels. Classes of errors were defined as predefined messages of a static analyzer, and graphs of the evolution of errors over time and for different subsystems were presented. Specifically, drivers have been found to be 3-7 times more error prone than other components.

Mutilin, Novikov and Khoroshilov made an analysis of typical errors in the drivers of the Linux operating system [4]. Here the concept of a typical error is introduced. According to the researchers, it is specific to a large number of drivers (for example, resource leaks, incorrect use of locks), while

a non-typical error is domain-specific for a particular driver. The authors manually analyzed the changes during the transition from one kernel version to another and compiled tables of the distribution of errors by classes. It was also found that drivers make 85% of all errors in the kernel. The paper by Novikov [5] continues this work, summarizes various statistics on changes in the kernel and concludes that about 40% of changes between stable versions of the kernel are fixes of typical errors. Since more versions were analyzed and the code evolved, the author had to supplement the previous created classes. Such manual analysis is more difficult, but the authors note that it is more careful.

Lus and Arpaci-Dusseaus manually analyzed 5079 patches related to file systems made over 8 years [6]. Classes of bugs, the so-called bug patterns, are identified and graphs of their evolution are given, as a result, a dataset of 1800 bugs is compiled.

The empirical work by Tan, Liu, Li, Wang, Zhou and Zhai [7] is devoted to a broad study of bugs in open-source software, including the Linux kernel. As their results for Linux, bugs are simply assigned to one of the subsystems (core, driver, network, FS, arch, other), while several open-source components are analyzed using message text from BugZilla with its vectorization and further automatic classification.

The work by Xiao, Zheng, Yin, Trivedi, Du and Cai [8] is devoted to the study of 5741 Linux kernel bug reports, which were analyzed according to the description, comments and attached files from the Linux kernel bug tracker [9]. Bugs are classified into fast-reproducible (Bohrbug), difficult-to-reproduce (Mandelbug) or context-dependent, and are also defined categories from which the bug context depends, that is, errors with memory, not freed resources, etc. At the same time, the authors built a network based on the Linux call graph, with the help of which they track the impact of the functions affected in bug reports by counting various metrics.

The researchers Melo, Flesborg, Brabrand and Wasowski present the results of compiling 42,060 kernels with all warnings enabled [10]. As a result of the analysis of 400,000 warnings, they classified by type and distribution by kernel subsystems and identified drivers as the most vulnerable portion of the kernel.

The work by Hoang, Lawall, Tian, Oentaryo and Lo [11] presents the PatchNet network, created as a result of automatic analysis of patches for the kernel, in order to predict whether a given patch will be accepted in the mainline kernel or not. For evaluation, the texts of the commit messages and the vector representation of the changes from the diff of the commit are used, which are then used to build a convolution neural network.

The research by Tian, Lawall and Lo [12] is separately devoted to determining whether a patch to the kernel is a bug fix or not. The authors note that simple analysis based on commit messages does not always lead to correct results and propose a model that uses two classification algorithms: Learning from Positive and Unlabeled Examples and Support Vector Machine. It also uses features extracted from the commit diff.

In the study by Acher, Martin, Pereira, Blouin, Khelladi and Jezequel [13], the authors provide infrastructure, classify and analyze Linux kernel-specific errors associated with errors in configuration files, as a rule, these are errors with dependencies. 95,854 Linux kernel builds were produced on random configurations, and of these, about 6% ended with errors, and which are discussed in the work. It is noted that the number of errors has decreased with previous findings, apparently due to testing processes with randomized configurations.

Summarizing the above on Linux bug analysis, it can be seen that (1) bugs in drivers are the most common; (2) different methods are used for classification, this is static analysis, build logs and patch analysis; (3) a lot of huge manual work has been done but the results may now be considered no longer relevant (the code is constantly changing). However, automatic classification by analyzing commits in git repositories has not been applied yet.

3. Preliminaries

To solve the problem of analyzing commit messages, we need to work with the Git repository at the program level. This means that from a program in one of the programming languages using some API, we need to get a list of commits, filter commits (by date, for example), iterate through them, obtain changes and the text of the commit message. Probably, the most famous C-library for this is *libgit2*. For JVM programs, the *JGit* library is popular. One can also use *EGit* to work with remote repositories, including access to pull requests data, but this is not necessary for the current project because we are working with the mainline kernel with changes already accepted. To be able to work with a git repository from code in Python, the *GitPython* library can be a good choice.

In order to obtain a set of fixing commits from a set of interesting commits, the easiest way to check the commit message to find a list of some signal words, but it is better to use some sort of AI-based commit classifier like the one discussed in the work [12].

The next step is finding the similar fixing commit messages in order to reveal the most common bug fixes. To compare commit messages, it is necessary to work out *fuzzy string matching* algorithms. Note that fuzzy string comparison is popular in bioinformatics.

Of practical interest are efficiently calculated string similarity measures, such as the Levenshtein distance. Formally, the Levenshtein distance $L(s_1, s_2)$ [14] between strings s_1 and s_2 can be calculated according to the following formulas:

$$\begin{aligned} L(s_1, s_2) := \quad & \forall\, i \in (0..|s_1|): d_{i,0} := i + 1; \\ & \forall\, j \in (0..|s_2|): d_{0,j} := j + 1; \\ & \forall\, i \in (1..|s_1|): \\ & \quad (\forall\, j \in (1..|s_2|): cost := (s_1[i-1] = s_2[j-1])\ ?\ 0 : 1 \\ & \quad d_{i,j} := \min(\min(d_{i-1,j} + 1, d_{i,j-1}), d_{i-1,j-1} + cost); \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad d_{|s1|,|s2|}. \end{aligned} \tag{1}$$

With it, to find the closest string to the existing ones, in the simplest implementation, one needs to calculate the distances between them using formula (1) and choose the minimum one. This method does not require preliminary preparation of strings and is susceptible to slight changes in them.

Our previous papers [15, 16] demonstrated that the use of the Levenshtein distance can provide a good understanding what is going on with the fixes in major Linux kernel parts. However, such an analysis for big repositories takes a lot of resources (we need to compare $O(\text{fixing commit count}^2)$ string comparisons) and its accuracy is very difficult to verify.

Therefore, in this paper, we would like to apply another simple method known from its use in search engines (the use of "bag of words" to convert a phrase into a vector + calculation of cosine similarity between vectors to further determine the minimal distance).

To calculate the distance between commits messages using the cosine similarity approach, it is required to represent the commit message string as a vector (from the features as the words of the message). Here we denote $w_{i,j}$ as the sign of the presence of the word j in the string i, while n specifies the number of words in the dictionary of unique words. Then we are able to calculate the cosine similarity between the vectors:

$$D(s_1, s_2) \coloneqq D((w_{1,1}, w_{1,2}, ..., w_{1,n}), (w_{2,1}, w_{2,2}, ..., w_{2,n})) = \frac{\sum_{i=1}^{n} w_{1,i} \cdot w_{2,i}}{\sqrt{\sum_{i=1}^{n} w_{1,i}^2 \cdot \sum_{i=1}^{n} w_{2,i}^2}} \tag{2}$$

In this case, permutations of words in a string will not change anything.

If we use dictionaries that give the stem word form for each word (without cases, endings, etc.), we can get rid of the problems of counting the same words in different components of vectors. The process is called lemmatization [17] or lemma normalization. In the simplest case, the Catvar

dictionary can be used [18]. Here, for each word from the commit message (column 1), its normalized form can be obtained (column 2), for example, here is a dictionary fragment for the words “fix”:

```
fix     fix $N$
fix     fix $V+0$
fixed   fix $V+ed$
fixed   fix $V+en$
fixes   fix $N+s$
fixes   fix $N+s$
fixes   fix $V+s$
fixing  fix $V+ing$
```

In more advanced cases, the StanfordCoreNLP API [19, 20] can be used. Since not only the stem, but also the part of speech is known for each word, when converting phrases into vectors, it is advisable to filter them, cutting off articles and frequently used words.

For the purposes of searching for strings with “strong components” or relevant words/tokens (i.e., to reduce the weights of frequently occurring words in a string), the tf-idf approach [21] can be applied. It does frequency counting, and with this, the vector components (features) instead of word appearance (1 or 0) will contain tf-idf weights. If we denote m_w as the number of occurrences of the word w into a commit message $m \in M$, and n_w as the total number of words in the document, and $|M|$ as the total number of messages, then:

$$tfidf(w,m,M) := tf(w,m) \times idf(w,M) = \frac{n_w}{\sum\limits_k n_k} \times \log \frac{|M|}{|\{m_i \in M \mid w \in d_i\}|} \quad (3)$$

If we are able to vectorize commit messages, then it makes sense to try to cluster them automatically. Clustering or cluster analysis involves the vectorization of given objects, calculating the distances between them according to a certain metric and dividing objects into clusters or groups of nearby objects. Vectorization involves the selection of key entities of objects and their presentation as a set of vectors of the same dimension. The clustering algorithm is a function $X \to Y$ that assigns a cluster identifier $y \in Y$ to any object $x \in X$. Some popular clustering algorithms are K-means, DBSCAN, and hierarchical clustering. The K-means algorithm iteratively minimizes the total square deviation of cluster points from the centers of these clusters (a classical approach presented in [22]). The density-based spatial clustering of applications with noise (DBSCAN) algorithm groups points in a high-density area into one cluster, while marking lonely points as noise [23]. With hierarchical clustering, a tree (dendrogram) is built, from leaves to root. Initially, each object is contained in its own cluster. Next, an iterative process of merging the two nearest clusters takes place until all clusters are combined into one, or the required number of clusters is found [24].

4. On the implementation

To work with git repositories, we utilize the Python git library, which enables us to iterate over commits and insert conditions to process them in the code. Initially, we began our solution as a JDK program since we had prior experience working with the JGit library. Subsequently, we implemented a prototype for analyzing a Thunderbolt repository (as described in the work-in-progress article [1]) by overriding classes for processing commit messages using the minimum Levenshtein distance, vectorization, and searching for minimum distances between phrases. Later on, we developed methods for clustering vectors from phrases. Currently, we use the Python language since it facilitates clustering methods with scipy packages.

The solution scheme is illustrated in *Fig. 1*. In this solution, we acquire all the commits of a given repository, filter them according to the dates of interest, and extract only those explicitly indicating a bug fix.

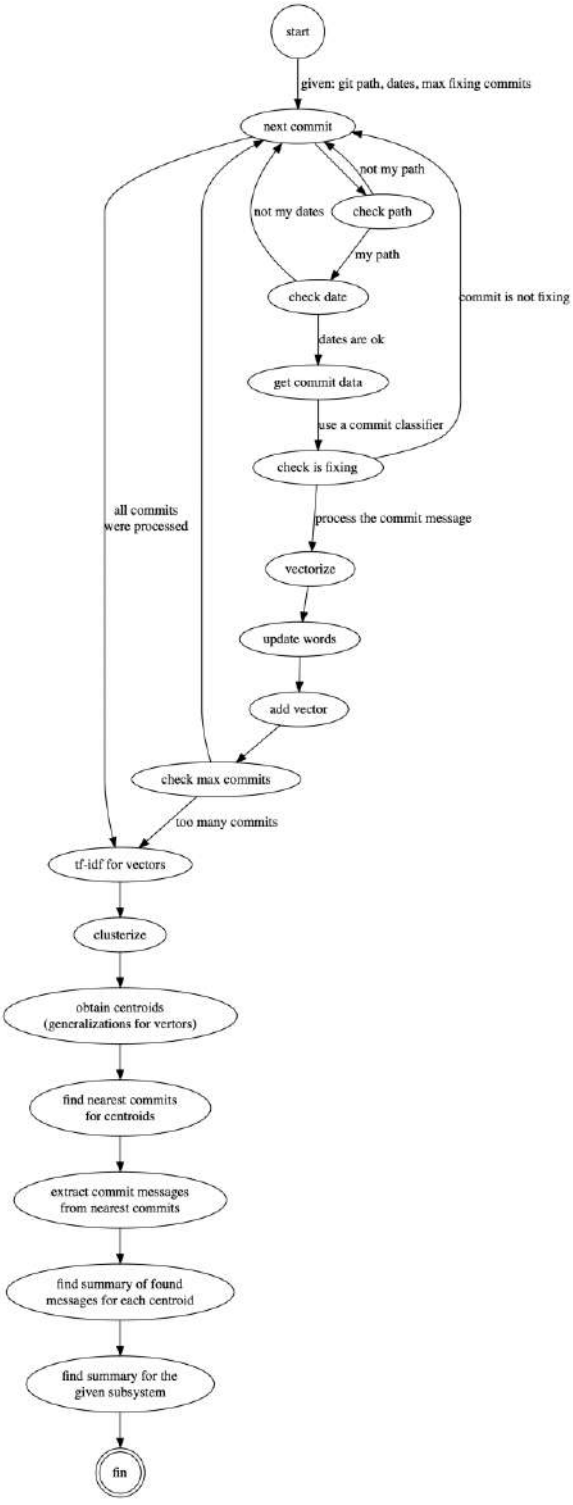


Fig. 1. A diagram of our solution

For this purpose, we employ our own implementation of the method discussed in the paper [12] using pre-trained classifiers based on information about the changes in the commit. Next, we work with vectorized and lemmatized fixing commit messages. We further form flat clusters from hierarchical clustering with a given threshold (which is calculated using some heuristics and can subsequently be improved by expert evaluation of the resulting clusters). We store the resulting vectors sorted by distance from the centroid, marking the most important words first. To obtain meaningful results, we need to find the nearest commits for each of the discovered vectors using the commit text. Afterward, through manual analysis, we can determine the most general message about the fix and an example of such a fix for the error in the code.

5. Findings

In this section, we provide the results of our analysis for the major subsystems of the Linux kernel. We analyzed some selected subsystems of the Linux kernel by examining the corresponding parts of the path in the main git repository. Due to limited computing resources, we were only able to process a maximum of 10,000 fixing commits for each subsystem. We present our findings in the form of a list of vectors, each representing a grouping (blurring) of the most frequent messages. For each vector, we provide a list of its components, sorted by importance, as well as git messages from the closest commits to that vector. They we analyze these messages in order to describe each vector in natural language. Finally, we provide a generalization of the fixes found for each analyzed subsystem.

5.1 Kernel (/linux/kernel)

Vector #1: [cpu, period, grace, rcu, event, callback, commit, probe, function, state, check, structure, rcu_node, stall, file]

```
Fix day-one dyntick-idle stall-warning bug
rcu: Suppress more involved false-positive preempted-task splats
rcu: Accelerate grace period if last non-dynticked CPU
rcu/segcblist: Prevent useless GP start if no CBs to accelerate
Go dyntick-idle more quickly if CPU has serviced current grace period
```

These fixes address the very important RCU subsystem in Linux, which provides ways to non-blockingly synchronize concurrent entities [25, 26]. However, there are problems in the form of potential unfinished waiting or inefficiency in its implementation, since processors in modern computing systems can go into energy-efficient hibernation, which leads to bugs in the RCU implementation for the tasks running on them (problems with the waiting period or grace period).

Vector #2: [buffer, kernel, ring, warning, doc, function, page, parameter, iterator, read, type, member, resource, trace, tracepoint]

```
ring-buffer: Fix kernel-doc
ring-buffer: Always reset iterator to reader page
resource/docs: Fix new kernel-doc warnings
seccomp: fix kernel-doc function name warning
rcu: Fix a kernel-doc warnings for "count"
```

These fixes refer to corrections to code documentation, which are done in the form of comments embedded in the code. The kernel-doc tool collects these comments and checks their completeness [27]. The comments here refer to the ring buffer, which can be used to implement efficient network applications [28].

Vector #3: [module, panic, patch, build, state, cpu, error, message, new, unloaded, code, function, kernel, notifier, list, load_module]

```
module: Ensure a module's state is set accordingly during module coming cleanup
code
livepatch: Fix subtle race with coming and going modules
debug: track and print last unloaded module in the oops trace
```

```
[PATCH] Kprobes: Reference count the modules when probed on it
debug: show being-loaded/being-unloaded indicator for modules
```

These fixes concern errors when loading and unloading kernel modules, more precisely during their live loading, when the already loaded code is replaced in a running system [29]. This is possible using the function tracing approach.

Vector #4: [tracer, function, graph, ret_stack, task, tracing, option, new, callback, code, ftrace, return, add, boot, buffer]

```
tracing/function-graph-tracer: drop the kernel_text_address check
function-graph: allow unregistering twice
tracing: Move mmio tracer config up with the other tracers
function-graph: move initialization of new tasks up in fork
tracing: Add ftrace events for graph tracer
```

These fixes concern the actual implementation of tracing [30] and working with the function call graph.

Vector #5: [timer, base, cpu, target, clk, idle, code, interval, posix, task, tick, jiffy, case, race, trace]

```
posix-timers: Fix full dynticks CPUs kick on timer rescheduling
timers: Use proper base migration in add_timer_on()
timer/trace: Improve timer tracing
posix-cpu-timers: Unbreak timer rearming
hrtimer: Preserve timer state in remove_hrtimer()
```

These fixes are aimed at fixing the kernel code for implementing timers according to the POSIX standard (see for example a discussion on its userspace interface [31]), namely errors during recharging (when changing the response time of already set timers), which entails working with related processes that may be located on temporarily retired processors.

Vector #6: [lock, ftrace, error, kernel, rlock, trace, deadlock, incompatible, type, possible, comparison, lockdep, info, irq, timekeeping]

```
timekeeping: Avoid possible deadlock from clock_was_set_delayed
sched/core: Make dl_b->lock IRQ safe
timekeeping: Fix HRTICK related deadlock from ntp lock changes
cpu/hotplug: Drop the device lock on error
pid: fix lockdep deadlock warning due to ucount_lock
```

These fixes are related to the internal kernel elapsed time measurement subsystem [32, 33] and associated incorrect locking in the implementation.

Vector #7: [console, printk, boot, message, list, line, early, srcu, add, time, tracepoint, use, patch, problem, console_lock]

```
console: prevent registered consoles from dumping old kernel message over again
[PATCH] CON_CONSDEV bit not set correctly on last console
printk: don't prefer unsuited consoles on registration
Revert "printk: Block console kthreads when direct printing will be required"
console: allow to retain boot console via boot option keep_bootcon
```

This series of fixes is devoted to kernel diagnostic messages and their output via tty consoles.

Vector #8: [lockdep, lock, patch, time, code, cross, performance, release, run, second, boot, case, kernel, bug, counter]

```
lockdep: spin_lock_nest_lock(), checkpatch fixes
lockdep, bug: Exclude TAINT_FIRMWARE_WORKAROUND from disabling lockdep
lockdep: more robust lockdep_map init sequence
locking/lockdep: Add a boot parameter allowing unwind in cross-release and disable
it by default
tracing: use raw spinlocks for trace_vprintk
```

These fixes are related to the work of the lockdep deadlock prevention tool [34] in the kernel and the work of the checkpatch tool [35] to check the formal requirements of the patches associated with it.

Vector #9: [kernel, inline, bpf, event, type, btf, pid, trace, number, buffer, foo, lock, rip, code, cat]

```
bpf: prevent decl_tag from being referenced in func_proto
tracing: Free buffers when a used dynamic event is removed
coredump: fix crash when umh is disabled
tracing: Fix memory leak in eprobe_register()
tracing: Check return value of __create_val_fields() before using its result
```

The fixes are related to BPF integration into the kernel and tracing (described in [36] and discussed in [37]), which were detected by the Syzkaller tool [38]. Since the tool reports contain listings with the same keywords (register dump, call stack), they were detected as similar vectors.

Vector #10: [error, return, code, function, value, failure, case, cgroup, file, negative, ret, add, userspace, bpf, caller]

```
cred: add missing return error code when set_cred_ucounts() failed
rcutorture: Fix error return code in rcu_perf_init()
bpf: Fix error return code in map_lookup_and_delete_elem()
ftrace: Deal with error return code of the ftrace_process_locs() function
genirq/timings: Fix error return code in irq_timings_test_irqs()
```

This series of fixes included fixes for the “fix error return code” error in various parts of the kernel, including the BPF and RCU torture functions [39].

In general, based on this key subsystem of the Linux kernel, we can conclude that most of the problems found and corrected were associated with incorrect operation of multiprocessor concurrent systems due to incorrect processing of all scenarios in the control flow, which involve accurate processing in conditions of variability of resources such as processors, pages memory, etc. That is, the handling of unexpected situations was not carried out completely correctly. The key kernel components mentioned were the RCU subsystem, swap management, timing, BPF and tracing. The code identified problems were related with the correct processing of return codes.

5.2 Drivers (/linux/drivers)

Fixes for kernel drivers are distinguished by the presence of a large number of identical changes (“serial patches”). Essentially, some change to the API is made and then the code for a large number of drivers that depend on that API should be changed. Such changes can be described in the form of so-called semantic patches and applied to a given set of files [40] and also attempted to be generalized from a set of source code files [41]. The next 10 vectors found describe exactly such changes, all of them are repeating.

Vector #1: [remove, return, function, null, check, unused, error, value, staging, is_err, macro, pointer, test, debug, definition]

```
mfd: pm8008: Fix return value check in pm8008_probe()
misc/pvpanic: fix return value check in pvpanic_pci_probe()
drm/i915/selftests: Fix return value check in live_breadcrumbs_smoketest()
n64cart: fix return value check in n64cart_probe()
net: sparx5: fix return value check in sparx5_create_targets()
```

Vector #2: [error, code, return, negative, function, case, success, net, scsi, ethernet, drm, mtk_eth_soc, path, add, einval]

```
RDMA/srpt: Fix error return code in srpt_cm_req_recv()
HID: pidff: fix error return code in hid_pidff_init()
mmc: usdhi6rol0: fix error return code in usdhi6_probe()
mtd: mtd_oobtest: fix error return code in mtd_oobtest_init()
net: sparx5: fix error return code in sparx5_register_notifier_blocks()
```

Vector #3: [dev_err, error, redundant, remove, message, print, devm_ioremap_resource, avoid, function, platform_get_irq, drivers, unnecessary, line, coccicheck, follow]

```
can: ctucanfd: Remove redundant dev_err call
fbdev: imxfb: Remove redundant dev_err() call
crypto: aspeed - Remove redundant dev_err call
mailbox: arm_mhu_db: Remove redundant dev_err call in mhu_db_probe()
soc/tegra: cbb: Remove redundant dev_err call
```

Vector #4: [array, member, flexible, element, replace, length, zero, struct, help, kernel, code, helper, use]

```
scsi: smartpqi: Replace one-element array with flexible-array member
staging: r8188eu: Replace zero-length array with flexible-array member
staging: rtl8723bs: Replace zero-length array with flexible-array member
scsi: megaraid_sas: Replace one-element array with flexible-array member in
MR_PD_CFG_SEQ_NUM_SYNC
scsi: megaraid_sas: Replace one-element array with flexible-array member in
MR_FW_RAID_MAP
```

Vector #5: [irq, interrupt, code, dt, core, hierarchical, resource, setup, static, use, platform_get_irq, allocation, cause, chaining, domain]

```
ata: pata_pxa: Use platform_get_irq() to get the interrupt
can: ti_hecc: ti_hecc_probe(): use platform_get_irq() to get the interrupt
serial: 8250_bcm7271: Use platform_get_irq() to get the interrupt
net: pxa168_eth: Use platform_get_irq() to get the interrupt
i2c: riic: Use platform_get_irq() to get the interrupt
```

Vector #6: [pm_runtime_resume_and_get, error, pm, counter, usage, order, runtime, use, decrement, add, medium, commit, usage_count, deal, dev]

```
media: i2c: ov9734: use pm_runtime_resume_and_get()
media: i2c: ov5675: use pm_runtime_resume_and_get()
media: i2c: ov5647: use pm_runtime_resume_and_get()
media: i2c: hi556: use pm_runtime_resume_and_get()
media: i2c: dw9807-vcm: use pm_runtime_resume_and_get()
```

Vector #7: [dev_err_probe, error, code, probe, use, check, dev_err, path, helper, print, debugfs, defer, reason, switch, replace]

```
usb: usb251xb: Switch to use dev_err_probe() helper
drm/panel: simple: Use dev_err_probe() to simplify code
backlight: ktd253: Switch to use dev_err_probe() helper
USB: PHY: JZ4770: Switch to use dev_err_probe() helper
usb: phy: generic: Switch to use dev_err_probe() helper
```

Vector #8: [line, blank, comment, checkpatch, declaration, issue, style, patch, staging, net, block, add, error, parenthesis]

```
net: c101: add blank line after declarations
net: hdlc_cisco: add blank line after declaration
net: hdlc: add blank line after declarations
net: sealevel: add blank line after declarations
net: pci200syn: add blank line after declarations
```

Vector #9: [return, error, i2c, remove, message, callback, make, value, void, core, device, preparation, patch, result, cleanup]

```
iio:light:tsl2583: Remove duplicated error reporting in .remove()
iio:light:isl29028: Remove duplicated error reporting in .remove()
iio:accel:mc3230: Remove duplicated error reporting in .remove()
iio:light:opt3001: Remove duplicated error reporting in .remove()
iio:light:jsa1212: Remove duplicated error reporting in .remove()
```

Vector #10: [dev_err_probe, use, error, code, benefit, deal, debugfs, defer, devices_deferred, file, function, help, helper, issue, make]

```
iio: st_lsm9ds0: Make use of the helper function dev_err_probe()
iio: st_sensors: Make use of the helper function dev_err_probe()
drm/panel: y030xx067a: Make use of the helper function dev_err_probe()
drm/panel: xpp055c272: Make use of the helper function dev_err_probe()
drm/panel: sofef00: Make use of the helper function dev_err_probe()
```

5.3 Memory management (/linux/mm)

Vector #1: [page, swap, entry, pte, mm, dirty, cache, pmd, patch, error, code, check, bit, fault, lock]

```
mm: fix data corruption caused by lazyfree page
mm, swap: Fix a race in free_swap_and_cache()
mm: filemap: coding style cleanup for filemap_map_pmd()
mm: invalidate hwpoison page cache page in fault path
mm, swap: fix swapoff with KSM pages
```

The most common error fixes in the memory subsystem are errors related to paging, including disabling swap for KSM pages [42], accessing one page from two processes at the same time, attempting to free the same page at a time; use a hardware-poisoned page remain in the page cache, leading to data corruption, as well as problems with proper handling of shared pages when using swapoff.

Vector #2: [slab, object, kmemleak, size, patch, kfence, slub, partial, add, debug, mm, kernel, allocation, list, lock]

```
SLUB: ensure that the number of objects per slab stays low for high orders
slub: Fix calculation of cpu slabs
slub: When allocating a new slab also prep the first object
slab, slub: remove size disparity on debug kernel
kfence: add sysfs interface to disable kfence for selected slabs.
```

These fixes are devoted to correcting errors in the implementation of the SLUB allocation subsystem. Such allocators are discussed in the presentation [43]. Some fixes were initiated after using the kmemleak tool [44]. The identified fixes relate to working with slab objects that the allocator operates on. The fix also concerns the implementation of the kfence tool [45], allowing users to selectively disable kfence for certain slabs to improve performance.

Vector #3: [page, thp, migration, error, mm, memory, check, fault, cow, hugetlb, process, issue, reference, anonymous, swapcache]

```
mm/memory-failure.c: transfer page count from head page to tail page after split
thp
mm: Wait for THP migrations to complete during NUMA hinting faults
mm: optimize do_wp_page() for exclusive pages in the swapcache
mm/huge_memory: streamline COW logic in do_huge_pmd_wp_page()
mm/rmap: fix missing swap_free() in try_to_unmap() after arch_unmap_one() failed
```

These fixes are devoted to the code for operating with transparent huge page (THP) [46], namely, counting the number of pages when a huge page is divided into several small ones, errors while waiting for the completion of migrations of such pages, incorrect copy-on-write behavior, as well as incorrect handling of errors when making unmap.

Vector #4: [damon, user, monitoring, patch, interface, space, context, region, scheme, mm, address, sysfs, file, support, debugfs]

```
mm/damon/core: allow non-exclusive DAMON start/stop
mm/damon/core: add a new callback for watermarks checks
mm/damon/core: add a callback for scheme target regions check
mm/damon/core: add a function for damon_operations registration checks
```

```
mm/damon/vaddr: register a damon_operations for fixed virtual address ranges
monitoring
```

The listed fixes concern expanding the functionality of the DAMON subsystem [47]. This subsystem provides a means of monitoring data in memory, and with its use, system developers can better understand data circulation and implement optimizations.

Vector #5: [compaction, scanner, patch, zone, page, free, migration, mm, pageblock, pfn, order, check, migrate, success, allocation]

```
mm, compaction: make whole_zone flag ignore cached scanner positions
mm, compaction: more robust check for scanners meeting
mm, compaction: more focused lru and pcplists draining
mm: compaction: reset cached scanner pfn's before reading them
mm: compaction: detect when scanners meet in isolate_freepages
```

These fixes are dedicated specifically to the compaction feature [48]. Compaction is a process that tries to reduce fragmentation in memory by grouping pages around. The fixes affect the processes of scanning such places using scanners that classify candidate pages as LRU (Least Recently Used) and PCP (Per-CPU Page). This is important because if two scanners meet, it means that the compaction process has reached a dead end and needs to be restarted. These fixes make the process more focused, reducing the amount of unnecessary work done during compaction. Also a fix is dealing with cached scanner page frame numbers (PFN), which are not being reset before being read.

Vector #6: [folio, convert, page, use, compound_head, mm, filemap, function, patch, byte, deactivate_page, head, caller, conversion, migrate]

```
migrate: convert unmap_and_move() to use folios
mm/memory: add vm_normal_folio()
mm/hugetlb: convert dissolve_free_huge_page() to folios
mm/damon: Convert damon_pa_young() to use a folio
mm/hugetlb: convert move_hugetlb_state() to folios
```

The fixes concern the functionality of functions for use folios instead of pages. Folios are a new data structure introduced in the kernel that represent a contiguous range of pages [49]. The use folios instead of pages can improve the efficiency of memory migration, moving huge pages or freeing huge pages.

Vector #7: [page, soft, offline, free, mm, compound, patch, hugetlb, hwpoison, tail, change, check, order, buddy, flag]

```
mm: check __PG_HWPOISON separately from PAGE_FLAGS_CHECK_AT_*
mm, hugetlb, soft_offline: save compound page order before page migration
mm/page_alloc: move pages to tail in move_to_free_list()
mm: verify compound order when freeing a page
mm,hwpoison: cleanup unused PageHuge() check
```

Current fixes are dedicated to improving the efficiency of page allocation by reducing fragmentation and improving cache locality, as well ensuring that the correct number of pages is freed to avoid issues with page splitting.

Vector #8: [memblock, memory, mm, static, make, add, list, array, free, region, reserved, allocator, bootmem, debug, expose]

```
mm: free memblock.memory in free_all_bootmem
mm/memblock: make memblock_remove_range() static
revert "mm/memblock: add missing include <linux/bootmem.h>”
mm/memblock: define memblock_physmem_add()
memblock: also dump physmem list within __memblock_dump_all
```

These fixes are dedicated to operate with memory blocks. These functions add a new range of physical memory to the memory block allocator, allowing for more efficient memory management.

In the fixes something is done to provide more information about the system memory layout and aid in debugging.

Vector #9: [doc, kernel, mm, function, parameter, warning, description, markup, error, add, shmem, slab, vmalloc, cleanup]

```
mm/mempolicy.c: parameter doc uniformization
mm/page_vma_mapped.c: add colon to fix kernel-doc markups error for check_pte
[PATCH] more kernel-doc cleanups, additions
mm: fix fatal kernel-doc error
mm: fix kernel-doc markups
```

This series of fixes is dedicated to documenting this subsystem using the kernel-doc annotations.

Vector #10: [kasan, tag, mode, based, memory, hardware, report, fault, kernel, boot, feature, patch, qemu]

```
kasan: simplify quarantine_put call site
mm/mm_init.c: report kasan-tag information stored in page->flags
kasan, kmemleak: reset tags when scanning block
kasan: add kasan mode messages when kasan init
kasan, arm64: move initialization message
```

These fixes are devoted to special tags when scanning a memory block in both the Kernel Address Sanitizer (KASAN) [50, 51] and Kernel Memory Leak (kmemleak) subsystems. This is done to provide more detailed information about memory usage and aid in debugging.

To summarize all these fixes, it can be noted that modern changes in the Linux memory subsystem relate to virtual memory, large memory blocks and various tools for memory monitoring and debugging. Working with memory is so complex that without additional tools that are provided by the kernel, debugging and optimization are not possible.

5.4 Scheduling (/linux/kernel/sched)

Vector #1: [deadline, dl_bw, task, total_bw, bw, dl_nr_running, dl_rq, sched, runtime, dl, parameter, sched_deadline, bandwidth, problem, value]

```
sched/deadline: Fix a bug in dl_overflow()
sched/debug: Add deadline scheduler bandwidth ratio to /proc/sched_debug
sched/deadline: Fix switching to -deadline
sched/deadline: Show leftover runtime and abs deadline in /proc/*/sched
sched/deadline: Fix migration of SCHED_DEADLINE tasks
```

A large number of fixes are dedicated to the Deadline Scheduler subsystem [52]. Namely, when tasks with a deadline exceeding the maximum value were not being correctly handled; when it is needed to add the ability to display the leftover runtime and absolute deadline for SCHED_DEADLINE tasks in the /proc/*/sched file, and also when tasks were not being correctly migrated between CPU cores.

Vector #2: [numa, node, fault, memory, rate, sched, task, mm, pte config_sched_debug, scan, access, commit, local]

```
sched/numa: Count pages on active node as local
mm: sched: numa: Delay PTE scanning until a task is scheduled on a new node
Revert "mm: sched: numa: Delay PTE scanning until a task is scheduled on a new
node"
mm: numa: Rate limit setting of pte_numa if node is saturated
sched/numa: Disable sched_numa_balancing on UMA systems
```

The following fixes are dedicated to scheduling in NUMA (Non-Uniform Memory Access) systems to improve performance and reduce memory latency. In particular, pages on the currently active node were not being counted as local; delays can be appear in scanning of page table entries (PTEs)

for NUMA balancing until a task is scheduled on a new node; pages on a saturated NUMA node could cause excessive memory usage; the scheduler NUMA balancing feature on Uniform Memory Access (UMA) systems, which do not have non-uniform memory access characteristics should be disabled.

Vector #3: [cpu, idle, load, task, active, online, domain, utilization, fair, balancing, nohz, numa, core, ilb, balance]

```
sched: Improve load balancing in the presence of idle CPUs
sched: Prevent raising SCHED_SOFTIRQ when CPU is !active
sched: Allow migrating kthreads into online but inactive CPUs
sched: Fix cpu_active_mask/cpu_online_mask race
sched/fair: Trigger the update of blocked load on newly idle cpu
```

The current fixes are devoted to the work of the scheduler when a CPU is idle. For example,to better handle situations where there are idle CPUs in the system; prevent the scheduler from raising the SCHED_SOFTIRQ interrupt on CPUs that are not currently active; allow kernel threads (kthreads) to be migrated to CPUs that are online but currently inactive; ensure that the scheduler updates the blocked load on CPUs that become idle. This can improve performance by making better use of available CPU resources.

Vector #4: [macro, sched, kernel, debug, check, commit, default, fair, field, magic, new, number, rename, runtime, schedstat_val]

```
sched: Move SCHED_LOAD_SHIFT macros to kernel/sched/sched.h
sched/debug: Fix /proc/sched_debug regression
sched/debug: Rename 'schedstat_val()' -> 'schedstat_val_or_zero()'
sched: Fix kernel-doc warnings in kernel/sched/fair.c
sched: Move wait.c into kernel/sched/
```

These fixes are about reorganizing the code.

Vector #5: [load, cpu, task, group, cfs_rq, entity, weight, cgroup, vruntime, fair, time, sched, cpuset, real, share]

```
sched/fair: Fix unfairness caused by missing load decay
sched/fair: Fix incorrect task group ->load_avg
sched: Avoid scale real weight down to zero
sched/fair: Remove task and group entity load when they are dead
sched: Aggregate total task_group load
```

The presented fixes usually relate to errors in the implementation of Completely Fair Scheduler (CFS) [53] and to task group load, which could lead to unfairness in the distribution of CPU resources. The design of CFS leads to the concept of schedulable entities, where tasks are managed by the scheduler as a whole. The fixes address the following situations: when the scheduler was not properly decaying the load of tasks over time, when it was calculating the load average of task groups incorrectly, when it was not properly removing the load of dead tasks and task groups, as well us there is some fix improves the way the scheduler aggregates the total load of task groups.

Vector #6: [sched, kernel, declaration, error, function, core, asm, implicit, paravirt, cputime, include, print_cfs_rq, werror, change, commit]

```
sched/headers: Prepare header dependency changes, move the <asm/paravirt.h>
include to kernel/sched/sched.h
sched/s390: Fix compile error in sched/core.c
sched/debug: Move print_cfs_rq() declaration to kernel/sched/sched.h
sched/core: Fix compilation error when cgroup not selected
sched/debug: Move the print_rt_rq() and print_dl_rq() declarations to…
```

These fixes are also about reorganizing the code.

Vector #7: [cpu, kernel, sched_domain, time, commit, flag, foundation, infradead, link, linux]

```
sched/isolcpus: Fix "isolcpus=" boot parameter handling when !CONFIG_CPUMASK_OFFSTACK
sched: Fix the broken sched_rr_get_interval()
sched/cputime: Resync steal time when guest & host lose sync
sched/nohz: Fix overflow error in scheduler_tick_max_deferment()
sched: Fix init NOHZ_IDLE flag
```

Current fixes match with scheduler domains [54] from different scheduling subsystems, including a bug that was not returning the correct time quantum for round-robin scheduling; a bug with incorrect CPU isolation; and an issue that steal time of a virtual CPU in a guest operating system could become out of sync with the host operating system.

Vector #8: [rq, task, cpu, curr, deadline, test, enqueue_task_dl, run, dl, hotplug, lock, sched, stress, offline, wa]

```
sched/deadline: Fix bad accounting of nr_running
sched/rt: Fix task stack corruption under __ARCH_WANT_INTERRUPTS_ON_CTXSW
sched/deadline: Add missing update_rq_clock() in dl_task_timer()
sched: Add missing rcu protection to wake_up_all_idle_cpus
sched/deadline: Fix the intention to re-evalute tick dependency for offline CPU
```

The next series of fixes concerns the work of deadline and real-time schedulers leading to incorrect scheduling decisions and system crashes: a counter for tasks and a clock value in a run queue were not properly updated; the task stack could become corrupted when interrupts were enabled during a context switch; a function was not properly protected by RCU.

Vector #9: [rt, task, cpu, current, run, user, tick, deadline, priority, pull, value, issue, sched, signal, time]

```
sched/rt: Avoid updating RT entry timeout twice within one tick period
sched/rt: Kick RT bandwidth timer immediately on start up
sched/rt: Do not pull from current CPU if only one CPU to pull
sched/rt: Fix RT utilization tracking during policy change
sched,cgroup: Fix up task_groups list
```

These fixes relate to RT (real-time) type schedulers for control groups [55, 56]. The latter concept defines a set of tasks, and all their future children, as a hierarchical group with specialized behavior. In details, the fixes address an issue where the real-time scheduler was updating the timeout value for a task twice within a single tick period; correct an issue where the real-time scheduler was delaying the start of the bandwidth timer; improve the behavior when the real-time scheduler was unnecessarily pulling tasks from the current CPU even when there was only one other CPU available, and fix the situation where the task_groups list was not properly updated when tasks were moved between cgroups, which lead to unnecessary overhead and degraded performance.

Vector #10: [update_rq_clock, add, core, sched, address, bug, clock, rq, update, double, effort, minimize, post_init_entity_util_avg, way]

```
sched/core: Add missing update_rq_clock() call in set_user_nice()
sched/core: Add missing update_rq_clock() in post_init_entity_util_avg()
sched/core: Add missing update_rq_clock() in detach_task_cfs_rq()
sched/core: Fix double update_rq_clock) calls in attach_task()/detach_task()
```

These fixes solve the same problem: lack or excess of function calls to update time data in the run queue per cpu structure of the scheduler.

To summarize, the fixes in the scheduling subsystem are mainly devoted to improving deadline, fair and realtime scheduling for groups, correct work with idle cpu state, calculation of deadlines and updating time counters, and work in NUMA systems.

5.5 Network (/linux/net)

Vector #1: [error, return, code, null, rate, function, check, value, case, net, use, icmp, mac80211, pointer, netfilter]

```
Bluetooth: fix error return code in rfcomm_add_listener()
sctp: fix error return code in __sctp_connect()
ieee802154: fix error return code in ieee802154_add_iface()
ah6: fix error return code in ah6_input()
netfilter: nf_conntrack: fix error return code
```

A large number of fixes concern returning error codes from functions. Such fixes were made for many parts of the network subsystem.

Vector #2: [net, inline, fb, mm, kasan, fc, kernel, include, socket, common, core, arch, linux, ipv6, x86]

```
net: sched: fix race condition in qdisc_graft()
ipv6: Fix KASAN: slab-out-of-bounds Read in fib6_nh_flush_exceptions
net: igmp: respect RCU rules in ip_mc_source() and ip_mc_msfilter()
tcp: cdg: allow tcp_cdg_release() to be called multiple times
devlink: Fix use-after-free when destroying health reporters
```

These fixes are related to typical errors in the code in this case for processing network things: a race condition that could occur when multiple processes attempt to modify the same network queueing discipline; a memory access error in the IPv6 forwarding information base (FIB6) code; a violation of RCU synchronization rules in the Internet Group Management Protocol (IGMP) code; an issue with the TCP Congestion Detection and Avoidance (CDG) algorithm, which could cause crashes or other errors if its release function was called multiple times; and a memory management error in the devlink subsystem, which could cause a "use-after-free" error if a health reporter object was destroyed while still in use.

Vector #3: [tcp, variable, unused, error, udp, warning, remove, function, ipv4, patch, net, compile, ipv6, mac80211, packet]

```
net: ipv4: avoid unused variable warning for sysctl
[IPV4]: Fix "ipOutNoRoutes" counter error for TCP and UDP
mac80211: fix warning: unused variable invoke_tx_handlers
ipv4: ipconfig: avoid unused ic_proto_used symbol
[TCP]: TCP highspeed build error
```

These fixes exclusively concern warnings and errors when building code for implementing things related to TCP.

Vector #4: [rfkill, net, error, whitespace, state, input, issue, change, core, pointer, case, hardware, software, class, device]

```
net/rfkill/rfkill-input.c needs <linux/sched.h>
[NET] RFKILL: Fix whitespace errors.
rfkill-gpio: include linux/mod_devicetable.h
rfkill: copy the name into the rfkill struct
rfkill: allow to get the software rfkill state
```

These fixes encapsulate various fixes associated with the RFKILL subsystem [57]. RFKILL can be used to disable wireless communication on a device. This can include devices such as Wi-Fi and Bluetooth radios. The RFKILL subsystem in the kernel provides a unified interface for controlling these devices. rfkill-gpio is a driver for RFKILL devices that are controlled by GPIO pins [58]. It allows the kernel to control the state of the RFKILL device by toggling the GPIO pin.

Vector #5: [vlan, bridge, device, packet, add, error, hardware, port, driver, network, filtering, header, notification, address, state]

```
bridge: vlan: Prevent possible use-after-free
rtnetlink: catch -EOPNOTSUPP errors from ndo_bridge_getlink
```

```
[VLAN]: Fix hardware rx csum errors
bridge: add vlan filtering change for new bridged device
vlan: Enable software emulation for vlan accleration.
```

This series of fixes is dedicated to VLAN and bridge devices. In particular, an error occurred when a user tries to get information about a bridge device that does not support the requested operation. Another fix addresses an issue with hardware rx checksums in VLAN devices. Hardware rx checksums are used to verify the integrity of network packets, but some devices were reporting incorrect checksums. Some fix adds support for VLAN filtering on new bridged devices. VLAN filtering is used to separate network traffic into different virtual LANs. And a fix enables software-based VLAN acceleration, ensuring that these systems can benefit from improved network performance. VLAN acceleration is used to improve network performance by offloading some of the processing required for VLAN filtering to hardware, but not all systems have this hardware capability.

Vector #6: [command, hci, error, typo, request, event, nfc, support, code, patch, net, add, framework, hci_req_add send]

```
Bluetooth: Fix HCI request framework
Bluetooth: HCI request error handling
Bluetooth: Reorganize set_connectable HCI command sending
Bluetooth: Add support for custom event terminated commands
NFC: Implement HCI driver or internal error management
```

These fixes are dedicated to Bluetooth and HCI framework [59]. Previously, we already realized that fixes for bluetooth are often made in the Linux kernel [15, 16]. The HCI request framework is responsible for sending commands and receiving responses between the host and the Bluetooth controller. This fix corrects the implementation of the framework, ensuring that commands are sent and received correctly.

Vector #7: [static, address, br, edr, discovery, controller, function, make, le, mode, command, patch, type, dual, setting]

```
Bluetooth: Introduce controller setting information for static address
Bluetooth: Support static address when BR/EDR has been disabled
Bluetooth: Fix issue with switching BR/EDR back on when disabled
Bluetooth: Check capabilities in BR/EDR and LE-Only discovery
Bluetooth: Fix advertising data flags with disabled BR/EDR
```

This fixes are devoted to a feature in the Bluetooth subsystem of the Linux kernel that allows users to set a static Bluetooth address for their device's Bluetooth controller. Previously, the Bluetooth address was generated randomly each time the device was restarted, which could cause issues with some Bluetooth devices that relied on a consistent address. With this fix, users can now set a static address that will remain the same across reboots. Then, the static Bluetooth address set in the previous fix is still used even if the device's Bluetooth controller has been disabled for BR/EDR (Basic Rate/Enhanced Data Rate) communication. Previously, if BR/EDR was disabled, the Bluetooth subsystem would generate a new random address each time the controller was enabled again.

Vector #8: [net, refcount, add, netpoll, tracker, device, struct, error, netns, socket, core, leak, dev_put, help, kernel]

```
net: add net device refcount tracker to struct pneigh_entry
netpoll: add net device refcount tracker to struct netpoll
net: add net device refcount tracker to struct netdev_rx_queue
net: add net device refcount tracker to struct netdev_adjacent
net: bridge: add net device refcount tracker
```

The mentioned fixes in the Linux kernel are related to adding a net device refcount tracker to different structures within the network subsystem. The reference count tracker helps to keep track

of the number of references to this structure, which is useful for ensuring accurate and reliable management of the network devices.

Vector #9: [unlock, error, path, mac80211, function, return, double, wifi, case, commit, deflink, lock, mlme, net, netfilter]

```
ceph: unlock on error in ceph_osdc_start_request()
mac80211: unlock on error path in ieee80211_ibss_join()
wifi: mac80211: unlock on error in ieee80211_can_powered_addr_change()
Bluetooth: delete a stray unlock
tipc: unlock in error path
```

These fixes check that any locks held are released before returning from a specified function even where an error is appeared. For example, in the Transparent Inter-Process Communication (TIPC) protocol implementation, this fix ensures that if an error occurs during specific operations, any locks held during the operation are released. This can prevent potential resource leaks.

Vector #10: [port, devlink, switch, cpu, dsa, user, case, driver, net, link, number, upstream, mtu, sw1p4, attribute]

```
net: dsa: give preference to local CPU ports
net: dsa: Fix type was not set for devlink port
devlink: append split port number to the port name
net: dsa: calculate the largest_mtu across all ports in the tree
netfilter: nf_ct_h323: fix bug in rtcp natting
```

These fixes are devoted to issues with network ports and improve the network performance on systems using Distributed Switch Architecture (DSA) [60]. DSA is a framework in Linux kernel that allows network switches to be controlled by the kernel. For example, with a fix, the kernel gives preference to the local CPU (Central Processing Unit) ports, which reduces latency and improves packet processing efficiency. Another fix addresses a bug related to the Maximum Transmission Unit (MTU) calculation in the Distributed Switch Architecture. And there is a fix to resolve a bug in the nf_ct_h323 module, which is responsible for Network Address Translation (NAT) handling for the H.323 protocol in netfilter (Linux network packet filtering subsystem). The bug specifically relates to Real-Time Control Protocol (RTCP) packets and their translation during NAT.

5.6 IRQ (/linux/kernel/irq)

Vector #1: [function, cleanup, kernel documentation parameter comment genirq irq patch recent update add addition commit core]

```
genirq: Fix handle_bad_irq kerneldoc comment
[PATCH] more kernel-doc cleanups, additions
[PATCH] genirq: cleanup: no_irq_type cleanups
Update __irq_domain_alloc_fwnode() function documentation
cpumask: Cleanup more uses of CPU_MASK and NODE_MASK
```

A large series of fixes in the IRQ subsystem is devoted to documentation, comments and code organization.

Vector #2: [pointer, null, function, check, alias, bug, crash, debugfs, genirq, handle, boot, compilation, dereference, driver, irqdomain]

```
genirq/irqdomain: Check pointer in irq_domain_alloc_irqs_hierarchy()
genirq/debugfs: Remove redundant NULL pointer check
genirq: Fix null pointer reference in irq_set_affinity_hint()
sparseirq: work around __weak alias bug
irqdomain: Fix debugfs formatting
```

Another series of fixes is devoted to errors in working with pointers.

Vector #3: [affinity, node, irq, interrupt, domain, genirq, software, case, device, driver, irqdomain, setup, allocation, bad, default]

```
genirq: Respect NUMA node affinity in setup_irq_irq affinity()
genirq/affinity: Fix node generation from cpumask
genirq: Move initial affinity setup to irq_startup()
irqdomain: Allow software nodes for IRQ domain creation
genirq/irqdomain: Add an irq_create_mapping_affinity() function
```

The main domain fixes in the IRQ subsystem relate to working with IRQ affinity [61]. That specifies which target CPUs are permitted for a given IRQ source. For example, a fix ensures that interrupts are assigned to the correct NUMA node based on the affinity settings, improving performance and reducing latency. Another fix corrects an issue where the node generation from a cpumask was not working correctly, leading to incorrect affinity settings. The next fix improves the initialization of interrupt affinities by moving it to the irq_startup() function, ensuring that affinities are set correctly from the beginning. Some other fix allows the creation of software nodes for interrupt domains, improving flexibility and allowing for better management of interrupts. Finally, a fix adds a new function to create interrupt mappings with specific affinity settings, allowing for more fine-grained control over interrupt allocation.

Vector #4: [gpio, error, drivers, ko, export, irq, add, function, chip, commit, export_symbol_gpl, following, generic, genirq]

```
irq: Add EXPORT_SYMBOL_GPL to function of irq generic-chip
device property: export irqchip_fwnode_ops
genirq: Add missing irq_to_desc export for CONFIG_SPARSE_IRQ=n
irq: Export handle_fasteoi_irq
```

The following fixes concern exporting some IRQ functionality so that it is available to other parts of the kernel, enabling them to interact with the irq chip functionality. For example, a fix adds the missing export statement for irq_to_desc, ensuring that it is accessible to other kernel components even when CONFIG_SPARSE_IRQ is disabled. Another fix exports the handle_fasteoi_irq symbol from the irq subsystem. handle_fasteoi_irq is a function that handles FastEOI (Fast End Of Interrupt) interrupts, which is a type of interrupt management mechanism [61].

Vector #5: [state, flag, interrupt, disabled, genirq, hardware, irq_data, add, callback, force, suspend, access, avoid, certain, change]

```
genirq/PM: Properly pretend disabled state when force resuming interrupts
genirq: Add irq disabled flag to irq_data state
genirq: Reflect IRQ_MOVE_PCNTXT in irq_data state
```

The latest series of big fixes relates to IRQ state management. A fix addresses a problem related to resuming interrupts after they have been disabled. In certain scenarios, interrupts may be forcibly resumed even if they were disabled. However, the interrupt controller may still maintain some internal state indicating that the interrupts are disabled. This fix ensures that the proper disabled state is correctly reflected when interrupts are forcefully resumed. In a next fix, an additional flag for the irq_data state is added to indicate whether the interrupts are disabled. This enables efficient checking of the disabled state, as well as simplifying the interrupt handling code. Another fix enhances the implementation of the IRQ_MOVE_PCNTXT feature in the genirq subsystem. IRQ_MOVE_PCNTXT is used to move an interrupt context from one CPU to another.

As a result, we can say that the main fixes for IRQ in recent years have been devoted to code refactoring, improving work with IRQ affinity, providing some functions for export to other subsystems, and improving IRQ state management to support disabling interrupts and interrupt context transfer to other processors.

5.7 x86 (/linux/arch/x86)

Vector #1: [xen, x86, microcode, error, arch, function, kernel, declaration, build, implicit, cpu, acpi, include, page, guest]

```
arch/x86/xen/suspend.c: include xen/xen.h
```

```
x86, xen: fix hardirq.h merge fallout
xen/tracing: fix compile errors when tracing is disabled.
xen/trace: Fix compile error when CONFIG_XEN_PRIVILEGED_GUEST is not set
xen: Move xen_setup_callback_vector() definition to include/xen/hvm.h
```

A large number of fixes for x86 are devoted to various aspects of implementing support for the Xen hypervisor [62] in the Linux kernel.

Vector #2: [iommu, tlb, flush, amd, x86, gart, add, aperture, device, kvm, vpid, patch, code, function, l2]

```
AMD IOMMU: add stats counter for domain tlb flushes
AMD IOMMU: add stats counter for single iommu domain tlb flushes
amd-iommu: add function to flush tlb for all devices
x86, AMD IOMMU: add detect code for AMD IOMMU hardware
x86, AMD IOMMU: add dma_ops initialization function
```

Another large number of fixes are devoted to the implementation of AMD IOMMU and working with TLB. IOMMU (Input-Output Memory Management Unit) is a hardware component that allows for virtualization of devices and memory management. TLB (Translation Lookaside Buffer, see the discussion in the tutorial [63]) is a cache used by the processor to store recently accessed memory translations. The TLB is used to speed up memory access. For example, a fix adds a statistics counter to track the number of times the IOMMU domain TLB is flushed. This improves performance monitoring and helps identify potential issues. Another fix adds detection code for AMD IOMMU hardware, allowing the kernel to properly identify and utilize the hardware.

Vector #3: [x86, vector, kernel, linux, redhat, boot, apic, intel, interrupt, code, cpu]

```
x86/entry/64: Clear registers for exceptions/interrupts, to reduce speculation
attack surface
x86/entry/64/compat: Clear registers for compat syscalls, to reduce speculation
attack surface
x86/traps: Fix up general protection faults caused by UMIP
x86, kasan, ftrace: Put APIC interrupt handlers into .irqentry.text
x86/kconfig/32: Rename CONFIG_VM86 and default it to 'n'
```

The set of updates focuses entirely on fixing known vulnerabilities. The first fix targets a security vulnerability called "speculation attack," which allows attackers to exploit modern processors' speculative execution feature to access sensitive data. This fix clears registers used during exceptions and interrupts to prevent unauthorized access. The second fix is similar, but it specifically addresses compatibility syscalls in 64-bit mode. These syscalls are used to run 32-bit applications on a 64-bit system, and the fix clears registers used during these syscalls to prevent exploitation. The third fix addresses a bug where UMIP was causing GP faults in certain situations, which could be exploited by attackers. UMIP is a security feature that prevents certain instructions from being executed in user mode. The fourth fix moves the APIC interrupt handlers to a more secure location in memory to prevent exploitation. Finally, the last fix renames a kernel configuration option and sets its default value to "no" to prevent attackers from exploiting the VM86 feature, which allows 16-bit applications to run in virtual 8086 mode. This feature is no longer necessary in modern systems and can be exploited by attackers to gain access to sensitive information.

Vector #4: [page, table, error, x86, code, k8, powernow, kvm, cpumask, cleanup, fault, struct, guest, address, bit]

```
KVM: SVM: Limit PFERR_NESTED_GUEST_PAGE error_code check to L1 guest
x86/espfix/xen: Fix allocation of pages for paravirt page tables
kvm: svm: Add support for additional SVM NPF error codes
x86, vmi: put a missing paravirt_release_pmd in pgd_dtor
mm: add pt_mm to struct page
```

The next set of fixes focuses on correcting vulnerabilities in the KVM, Xen hypervisor, AMD Secure Virtual Machine, and Virtual Machine Interface subsystems. These fixes aim to prevent malicious

guest operating systems from crashing the host system and exploiting vulnerabilities to gain access to sensitive information. The updates include limiting error code checks for PFERR_NESTED_GUEST_PAGE to only the first level guest, correcting page allocation for paravirtualized page tables, adding support for additional error codes related to Nested Page Faults, and adding a new field to track memory management context in the struct page data structure.

Vector #5: [mmu, page, tdp, kvm, shadow, vcpu, sp, x86, guest, code, change, check, l1, use, table]

```
KVM: x86/mmu: Pivot on "TDP MMU enabled" to check if active MMU is TDP MMU
KVM: x86/mmu: Protect marking SPs unsync when using TDP MMU with spinlock
KVM: MMU: move mmu pages calculated out of mmu lock
KVM: x86: nSVM: fix switch to guest mmu
KVM: MMU: move the relevant mmu code to mmu.c
```

The KVM hypervisor also undergo changes to improve its performance and prevent data corruption. The changes include checking if TDP (two dimensional paging) MMU is enabled instead of checking the CPU model to ensure that the correct MMU is used for virtual machines running on different CPU models. The addition of spinlock protection when marking shadow page tables as unsynchronized in a TDP MMU environment prevents race conditions. Moving the calculation of MMU pages outside of the MMU lock reduces contention on the lock. Finally, a bug in the nested SVM code is corrected to ensure that the hypervisor switches to the guest MMU when running a nested virtual machine. Some links and an approach to formally verify such things are presented in the paper [64].

Vector #6: [reboot, x86, dell, pci, quirk, add, platform, acpi, kernel, power, show_bug, ce4100, id]

```
ACPI, x86: fix Dell M6600 ACPI reboot regression via DMI
x86/reboot: Add Zotac ZBOX CI327 nano PCI reboot quirk
x86/ce4100: Fix reboot by forcing the reboot method to be KBD
x86: Add Dell OptiPlex 760 reboot quirk
x86/reboot: Limit Dell Optiplex 990 quirk to early BIOS versions
```

These fixes address specific issues related to reboot functionality on certain computer systems by adding or limiting certain quirks in the kernel.

Vector #7: [bank, error, mce, cpu, x86, check, hardware, machine, number, value, amd, interrupt, type, register, disable]

```
x86/MCE: Determine MCA banks' init state properly
x86/mce: Avoid reading every machine check bank register twice.
x86/mce/AMD, EDAC/mce_amd: Enumerate Reserved SMCA bank type
x86/MCE: Initialize mce.bank in the case of a fatal error in mce_no_way_out()
x86/MCE/AMD: Define a function to get SMCA bank type
```

These fixes have been made to correct a bug in the Machine Check Architecture (MCA) code, optimize it by avoiding unnecessary reads of machine check bank registers, add support for enumerating reserved System Management Control Address (SMCA) bank types in the MCA code for AMD processors, and initialize the MCA bank variable in case of a fatal error in the MCA code. MCA is an internal architecture subsystem which detects and captures errors occurring within the microprocessor's logic [65].

Vector #8: [pci, bridge, window, e820, host, lenovo, region, space, x86, acpi, bar, address, device, resource, bus]

```
x86/PCI: Revert "x86/PCI: Clip only host bridge windows for E820 regions"
x86/PCI: Add kernel cmdline options to use/ignore E820 reserved regions
x86/PCI: Mark ATI SBx00 HPET BAR as IORESOURCE_PCI_FIXED
x86/PCI: Use host bridge _CRS info on Foxconn K8M890-8237A
x86/PCI: Ignore E820 reservations for bridge windows on newer systems
```

These fixes aim to improve the functionality of PCI devices in computer systems, in order to improve PCI device enumeration and resource allocation and ensure efficient use of available memory

resources (the so-called PCI quirks [66]). They include reverting a previous change that caused issues with certain devices accessing memory regions, adding kernel command line options for users to specify the use of reserved memory regions, marking the HPET BAR on ATI SBx00 chipsets as a fixed PCI resource, updating the PCI code for Foxconn K8M890-8237A systems, and modifying the PCI code to ignore reserved memory regions for bridge windows on newer systems.

Vector #9: [x86, style, problem, i8259, function, cleanup, impact, include, arch, asm, shutdown, code, error, file, irqinit_32]

```
x86: i8259.c fix style problems
x86: module_64.c fix style problems
x86: irq_32.c fix style problems
x86: time_32.c fix style problems
x86: irqinit_32.c fix style problems
```

Here are code style issues in x86-related components.

Vector #10: [bit, 32, 64, ptrace, x86, kernel, value, argument, code, syscall, firmware, mixed, mode, sign, signal]

```
x86_64: make ptrace always sign-extend orig_ax to 64 bits
x86/mpx: Fix 32-bit address space calculation
x86: ptrace: set TS_COMPAT when 32-bit ptrace sets orig_eax>=0
x86/efi: Truncate 64-bit values when calling 32-bit OutputString()
efi/libstub: Distinguish between native/mixed not 32/64 bit
```

The fixes for x86_64 and x86/efi are important to ensure proper handling of 32-bit applications running on a 64-bit system and to distinguish between native and mixed mode in the EFI libstub code accurately.

To sum up, fixes for this historical part of the kernel are devoted to code reorganization, addressing known types of vulnerabilities, better support for virtualization, playing around with quirks for specific hardware, as well as providing support for legacy systems and subsystems.

5.8 ARM64 (/linux/arch/arm64)

It turned out that a huge number of fixes for the ARM platform are devoted primarily to additions to device trees [67, 68]. There are a huge number of hardware configurations for the platform, and such configurations are described through a device tree in order to be accessible later programmatically by loading the appropriate drivers. Our software classified such fixes and the first 6 classes found describe support for different types of devices in the device tree.

Vector #1: [add, node, arm64, dts, renesas, support, device, thermal, tpu, patch, soc, zone, fd, hdmi, r8a77995]

```
arm64: dts: renesas: r8a77995: add thermal device support
arm64: dts: renesas: r8a774c0: Add thermal support
arm64: dts: renesas: r8a779a0: Add TPU device node
arm64: dts: renesas: r8a77990: add thermal device support
arm64: dts: renesas: r8a77965: Add SATA controller node
```

Vector #2: [clock, node, arm64, dts, add, pcie, controller, qcom, sdm845, ipa, phy, support, soc, dt, device]

```
arm64: dts: exynosautov9: add fsys0/1 clock DT nodes
arm64: dts: qcom: sdm845: add apr nodes
arm64: dts: rockchip: add the PCIe PHY for RK3399
arm64: dts: sdm845: Add lpasscc node
arm64: dts: qcom: sm6350: add IPA node
```

Vector #3: [dts, arm64, add, device, node, r8a7795, r8a7796, renesas, enable, salvator, hihope, support, usb3, board, patch]

```
arm64: dts: r8a7795: add usb2_phy device nodes
```

```
arm64: dts: r8a7795: Add USB-DMAC device nodes
arm64: dts: r8a7796: Add USB-DMAC device nodes
arm64: dts: renesas: r8a7795: add USB3.0 peripheral device node
arm64: dts: renesas: r8a7796: add USB3.0 peripheral device node
```

Vector #4: [meson, gxbb, arm64, dts, add, node, gxl, pin, usb, cec, enable, regulator, controller, amlogic, dt]

```
ARM64: dts: meson-gxbb: Add Meson GXBB PWM Controller nodes
ARM64: dts: meson-gxbb: Add CEC pins nodes
ARM64: dts: meson-gxbb-p20x: Enable USB Nodes
ARM64: dts: meson-gxbb-odroidc2: Enable USB Nodes
ARM64: dts: meson-gxbb-nexbox-a95x: Enable USB Nodes
```

Vector #5: [regulator, pwm, supply, usb, add, arm64, dts, vddcpu, device, power, dummy, gpu, boot, change, board, meson]

```
arm64: dts: meson-g12b: Fix the pwm regulator supply properties
arm64: dts: meson-sm1: Fix the pwm regulator supply properties
arm64: tegra: Fix Jetson Nano GPU regulator
arm64: dts: meson: odroid-c2: Add missing regulator linked to P5V0 regulator
arm64: dts: rockchip: Correct regulator for USB host on Odroid-Go2
```

Vector #6: [node, table, opp, soc, arm64, dts, bus, add, cpu_thermal, doe, property, build, cpu, following, qup]

```
arm64: dts: imx8mq: Move the opp table out of bus node
arm64: dts: ls1046a: Move cpu_thermal out of bus node
arm64: dts: ls1043a: Move cpu_thermal out of bus node
arm64: dts: ls1012a: Move cpu_thermal out of bus node
arm64: dts: qcom: sm8250: Move qup-opp-table out of soc node
```

Vector #7: [tcsr, mutex, address, space, mmio, device, arm64, dts, qcom, syscon, allow, check, dedicated, dt, halt]

```
arm64: dts: qcom: sdm845: switch TCSR mutex to MMIO
arm64: dts: qcom: msm8998: switch TCSR mutex to MMIO
arm64: dts: qcom: sm8150: switch TCSR mutex to MMIO
arm64: dts: qcom: qcs404: switch TCSR mutex to MMIO
arm64: dts: qcom: sc7180: switch TCSR mutex to MMIO
```

TCSR (Thread Context Save and Restore) is a feature in Qualcomm processors that provides a mechanism for saving and restoring the state of a thread during context switching. These fixes modify the implementation to use MMIO (Memory-Mapped Input/Output), which enables direct access to the TCSR register through memory addresses.

Vector #8: [vdso, arm64, offsets, kernel, support, arch, makefile, rule, enable, mremap, page, vma, bti, handling, orphan]

```
arm64: fix vdso-offsets.h dependency
arm64/vdso: Support mremap() for vDSO
arm64: vdso: Enable vDSO compat support
arm64: vdso: Map the vDSO text with guarded pages when built for BTI
arm64: vdso: enable orphan handling for VDSO
```

These fixes are devoted to the virtual dynamic shared object (vDSO) [69] implementation on arm64. vDSO provides a fast and efficient interface for certain system calls, allowing user-space programs to directly access kernel functionality without needing to transition to kernel mode. Enabling vDSO compatibility support ensures that both newer and older versions of user-space programs can make use of the vDSO. The mremap() system call is used to move memory mappings to a new address range or change their size. By enabling mremap() support for the vDSO, it allows for more efficient management of the vDSO memory mapping. A listed fix enhances the security of the vDSO on arm64 when built with support for Branch Target Identification (BTI). BTI is a security feature that

protects against certain control-flow attacks. Orphan handling refers to the ability to handle processes that have become detached or disassociated from their parent process. Enabling orphan handling for the vDSO ensures that it behaves correctly in scenarios where a process using the vDSO is orphaned.

Vector #9: [asm, arm64, function, arch, error, declaration, implicit, kernel, insn, mark, rutland, include, linux, bug]

```
arm64: fix missing asm/io.h include in kernel/smp_spin_table.c
arm64: fix missing asm/alternative.h include in kernel/module.c
arm64: fix missing asm/pgtable-hwdef.h include in asm/processor.h
arm64: fix missing linux/bug.h include in asm/arch_timer.h
arm64: kaslr: Use standard early random function
```

These fixes concern the reorganization of included files with headers for the ARM64 platform.

Vector #10: [sve, sme, state, flag, arm64, register, host, load, storage, vcpu, access, kvm, mode, context, el1]

```
arm64/sme: Don't flush SVE register state when allocating SME storage
arm64/signal: Clean up SVE/SME feature checking inconsistency
KVM: arm64: Move SVE state mapping at HYP to finalize-time
KVM: arm64: Trap host SVE accesses when the FPSIMD state is dirty
KVM: arm64: Always start with clearing SVE flag on load
```

Several fixes have been made to improve the handling and synchronization of the Scalable Vector Extension (SVE) state [70]. The first fix prevents unnecessary flushing of the SVE register state during Secure Memory Encryption (SME) storage allocation. The next fix resolves an inconsistency in feature checking for SVE and SME during signal processing, ensuring correct behavior. Moving the mapping of SVE state in Kernel-based Virtual Machine (KVM) to finalize-time ensures proper handling and synchronization. Host SVE accesses are now trapped in KVM when the Floating-Point SIMD (FPSIMD) state is dirty, preventing corruption. Lastly, consistently clearing the SVE flag during load operations in KVM guarantees correct initialization and handling of SVE features during virtualization, avoiding potential issues related to its state.

Thus, for ARM64, the defining fixes are the reorganization of the device tree set, reorganization of the code, synchronization of access to the state of internal registers and operations with virtual dynamic shared objects.

6. Conclusion

In this study, we utilized natural language messages from the Linux kernel git repository to identify the most common types of errors. Our approach involved clustering fixing commit messages to gain insight into the prevalent error classes in specific domains, such as memory management and scheduling subsystems. Unlike previous studies, we included links to specific commits, providing examples of bug reports for each generalized vector identified. This differed from our earlier approach, which relied on matching similar messages and working with frequently occurring messages but not similar generalization vectors.

Our analysis yielded satisfactory results, presenting the main vectors (in the form of keywords) and example messages for each class of fix. We also conducted a manual summary analysis for each class, revealing typical fixes for each subsystem and the automated tools used to detect them. We referenced these tools in our article and concluded that developers must study them when working with the corresponding subsystem.

It is important to note that the analysis conducted in this study is relatively straightforward and can be applied to any repository, with the ability to evaluate changes retrospectively. By selecting specific dates and parts of the repository, it is possible to identify errors and gain valuable insights into their prevalence. However, there is always room for improvement in any analysis, and our current implementation could benefit from enhancements such as refining the detection of fixing

commits, improving the commit coupling thresholds for hierarchical clustering, better normalization and removal of stop words, and developing methods for automatically describing each class of errors found. These improvements are crucial for ensuring the accuracy and effectiveness of our analysis, and we plan to address them in future research.

Ultimately, our goal is to provide system developers with a comprehensive understanding of the most common types of errors in various subsystems, along with the tools and techniques needed to address them effectively.

Acknowledgement

The work on this article was carried out at personal expense and in their own time by the authors.

We would like to express our gratitude to A.K. Petrenko and A. Kamkin for organizing the SYRCoSE colloquium, as well as A. Martyshkin and A. Vorontsov for conducting excursions around Penza and Nikolsk.

References

[1]. Starovoytov N., Golovnev N., Staroletov S. Towards methods to automatically identify the most common errors in Linux by analyzing git commit messages. In Proc. of the Spring/Summer Young Researchers' Colloquium on Software Engineering, 2023

[2]. Chou A., Yang, J. Chelf B., Hallem S., Engler D. An empirical study of operating systems errors. In Proc. of the eighteenth ACM symposium on Operating systems principles, 2001, pp. 73–88.

[3]. Palix N., Thomas G., Saha S., Calves C., Lawall J., Muller G. Faults in Linux: Ten years later. In Proc. of the sixteenth international conference on Architectural support for programming languages and operating systems, 2011, pp. 305–318.

[4]. Mutilin V., Novikov E., Khoroshilov A. Analysis of typical errors in Linux OS drivers (in Russian), Proceedings of the Institute for System Programming of the Russian Academy of Sciences, vol. 22, pp. 349–374, 2012.

[5]. Novikov E.M. Evolution of the Linux OS kernel (in Russian). Proceedings of the Institute for System Programming of the Russian Academy of Sciences, vol. 29, no. 2, pp. 77–96, 2017.

[6]. Lu L., Arpaci-Dusseau A.C., Arpaci-Dusseau R.H., Lu S. A study of Linux file system evolution. ACM Transactions on Storage (TOS), vol. 10, no. 1, pp. 1–32, 2014.

[7]. Tan L., Liu C., Li Z., Wang X., Zhou Y., Zhai C. Bug characteristics in open source software. Empirical software engineering, vol. 19, pp. 1665–1705, 2014.

[8]. Xiao G., Zheng Z., Yin B., Trivedi K.S., Du X., Cai K.-Y. An empirical study of fault triggers in the Linux operating system: An evolutionary perspective. IEEE Transactions on Reliability, vol. 68, no. 4, pp. 1356–1383, 2019.

[9]. Kernel.org. Bugzilla. Available at: https://bugzilla.kernel.org, accessed Sep. 14, 2023.

[10]. Melo J., Flesborg E., Brabrand C., Wasowski A. A quantitative analysis of variability warnings in Linux. In Proc. of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, 2016, pp. 3–8.

[11]. Hoang T., Lawall J., Tian Y., Oentaryo R.J., Lo D. PatchNet: Hierarchical deep learning-based stable patch identification for the Linux kernel. IEEE Transactions on Software Engineering, vol. 47, no. 11, pp. 2471–2486, 2019.

[12]. Tian Y., Lawall J., Lo D. Identifying Linux bug fixing patches. In Proc. of 2012 34th international conference on software engineering (ICSE). IEEE, 2012, pp. 386–396.

[13]. Acher M., Martin H., Pereira J.A., Blouin A., Khelladi D.E., Jezequel J.-M. Learning from thousands of build failures of Linux kernel configurations. Ph.D. dissertation, Inria; IRISA, 2019.

[14]. Levenshtein V.I. Binary codes with correction of dropouts, insertions and character substitutions (in Russian). Reports of the Academy of Sciences, vol. 163, no. 4. Russian Academy of Sciences, 1965, pp. 845–848.

[15]. Staroletov S. M. Researching the most common bugs in the Linux kernel by analysing commits in the git repository (in Russian). System Administrator, vol. 4(197), pp. 73–77, 2019 Available at: http://samag.ru/archive/article/3859, accessed Sep. 14, 2023.

[16]. Staroletov S. A survey of most common errors in Linux kernel. SYRCoSE Poster session, 2017.

[17]. Hann M. Towards an algorithmic methodology of lemmatization. Bulletin Association for Literary and Linguistic Computing, vol. 3, no. 2, pp. 140–150, 1975.
[18]. Categorial Variation Database (version 2.1). Available at: https://github.com/nizarhabash1/catvar, accessed Sep. 14, 2023.
[19]. Manning C.D., Surdeanu M., Bauer J., Finkel J. R., Bethard S., McClosky D. The Stanford CoreNLP natural language processing toolkit. In Proc. of 52nd annual meeting of the association for computational linguistics: system demonstrations, 2014, pp. 55–60.
[20]. Class StanfordCoreNLP. Available at: https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/pipeline/StanfordCoreNLP.html, accessed Sep. 14, 2023.
[21]. Salton G., Fox E. A., Wu H. Extended boolean information retrieval Communications of the ACM, vol. 26, no. 11, pp. 1022– 1036, 1983.
[22]. H. Steinhaus et al. Sur la division des corps mate´riels en parties. Bull. Acad. Polon. Sci, vol. 1, no. 804, p. 801, 1956.
[23]. Ester M., Kriegel H.-P., Sander J., Xu X. et al. A density-based algorithm for discovering clusters in large spatial databases with noise. KDD, vol. 96, no. 34, 1996, pp. 226–231.
[24]. Ward J.H. Hierarchical grouping to optimize an objective function. Journal of the American statistical association, vol. 58, no. 301, pp. 236–244, 1963.
[25]. McKenney P.E. A Tour Through TREE_RCU's Grace-Period Memory Ordering. Available at: https://www.kernel.org/doc/html/latest/RCU/Design/Memory-Ordering/Tree-RCU-Memory-Ordering.html, accessed Sep. 14, 2023.
[26]. McKenney P.E., Fernandes J., Boyd-Wickizer S., Walpole J. RCU usage in the Linux kernel: Eighteen years later. ACM SIGOPS Operating Systems Review, vol. 54, no 1, pp. 47-63, 2020.
[27]. Linux kernel guide. Writing kernel-doc comments. Available at: https://docs.kernel.org/doc-guide/kernel-doc.html, accessed Sep. 14, 2023.
[28]. Staroletov. S., Chudov R. An anomaly detection and network filtering system for Linux based on Kohonen maps and variable-order Markov chains. In Proc. Conference of Open Innovations Association, vol. 32, pp. 280-290, 2022. – EDN NNASCK.
[29]. Linux kernel guide. Livepatch. Available at: https://www.kernel.org/doc/html/latest/livepatch/livepatch.html, accessed Sep. 14, 2023.
[30]. Linux kernel guide. Ftrace - Function Tracer. Available at: https://www.kernel.org/doc/html/latest/trace/ftrace.html, accessed Sep. 14, 2023.
[31]. Linux manual page. timer_create(2). Available at: https://man7.org/linux/man-pages/man2/timer_create.2.html, accessed Sep. 14, 2023.
[32]. Linux kernel guide. ktime accessors. Available at: https://www.kernel.org/doc/html/latest/core-api/timekeeping.html, accessed Sep. 14, 2023.
[33]. S. Boyd. Timekeeping in the Linux kernel. Available at: https://elinux.org/images/0/0e/Timekeeping_in_the_Linux_Kernel_0.pdf, accessed Sep. 14, 2023.
[34]. Linux kernel guide. Runtime locking correctness validator. Available at: https://www.kernel.org/doc/html/latest/locking/lockdep-design.html, accessed Sep. 14, 2023.
[35]. Linux kernel guide. Checkpatch. Available at: https://www.kernel.org/doc/html/latest/dev-tools/checkpatch.html, accessed Sep. 14, 2023.
[36]. Linux kernel guide. BPF (Berkeley Packet Filter) Documentation. Available at: https://www.kernel.org/doc/html/latest/bpf/index.html, accessed Sep. 14, 2023.
[37]. Linux kernel guide. R. Davoli, M. Di Stefano, 2019. Berkeley Packet Filter: theory, practice and perspectives (Doctoral dissertation, Master's thesis, Universita di Bologna). Available at: https://amslaurea.unibo.it/19622/1/berkeleypacketfilter_distefano.pdf, accessed Sep. 14, 2023.
[38]. Google. Syzkaller. Available at: https://github.com/google/syzkaller, accessed Sep. 14, 2023.
[39]. Linux kernel guide. RCU Torture Test Operation. Available at: https://www.kernel.org/doc/html/latest/RCU/torture.html, accessed Sep. 14, 2023.
[40]. Lawall J., Muller G. Automating Program Transformation with Coccinelle. In Proc. of NASA Formal Methods Symposium, pp. 71-87, 2022.
[41]. Serrano L., Nguyen V.A., Thung F., Jiang L., Lo D., Lawall J., Muller G. SPINFER: Inferring Semantic Patches for the Linux Kernel. In Proc. of 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 235-248, 2020.
[42]. Linux kernel guide. Kernel Samepage Merging. Available at: https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html, accessed Sep. 14, 2023.

[43]. Lameter C. Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB. LinuxCon/Düsseldorf, 2014. Available at: https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf, accessed Sep. 14, 2023.
[44]. Linux kernel guide. Kernel Memory Leak Detector. Available at: https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html, accessed Sep. 14, 2023.
[45]. Linux kernel guide. Kernel Electric-Fence (KFENCE). Available at: https://docs.kernel.org/dev-tools/kfence.html, accessed Sep. 14, 2023.
[46]. Linux kernel guide. Transparent Hugepage Support. Available at: https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html, accessed Sep. 14, 2023.
[47]. Linux kernel guide. DAMON: Data Access MONitor. Available at: https://docs.kernel.org/mm/damon/index.html, accessed Sep. 14, 2023.
[48]. Linux kernel guide. Concepts overview. Available at: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html, accessed Sep. 14, 2023.
[49]. Corbet J. Clarifying memory management with page folios. Available at: https://lwn.net/Articles/849538/, accessed Sep. 14, 2023.
[50]. Linux kernel guide. The Kernel Address Sanitizer (KASAN). Available at: https://www.kernel.org/doc/html/latest/dev-tools/kasan.html, accessed Sep. 14, 2023.
[51]. Google. Kernel sanitizers. Available at: https://github.com/google/kernel-sanitizers, accessed Sep. 14, 2023.
[52]. Linux kernel guide. Deadline Task Scheduling. Available at: https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html, accessed Sep. 14, 2023.
[53]. Linux kernel guide. CFS Scheduler. Available at: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html, accessed Sep. 14, 2023.
[54]. Linux kernel guide. Scheduler Domains. Available at: https://www.kernel.org/doc/html/latest/scheduler/sched-domains.html, accessed Sep. 14, 2023.
[55]. Menage. P. Linux kernel guide. Control Groups. Available at: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html, accessed Sep. 14, 2023.
[56]. Linux kernel guide. Real-Time group scheduling. Available at: https://www.kernel.org/doc/html/latest/scheduler/sched-rt-group.html, accessed Sep. 14, 2023.
[57]. Linux kernel guide. rfkill - RF kill switch support. Available at: https://docs.kernel.org/driver-api/rfkill.html, accessed Sep. 14, 2023.
[58]. Linux kernel guide. Using GPIO Lines in Linux. Available at: https://www.kernel.org/doc/html/latest/driver-api/gpio/using-gpio.html, accessed Sep. 14, 2023.
[59]. Bluetooth. Core Specification 5.4, 2023. Available at: https://www.bluetooth.com/specifications/specs/core-specification-5-4/, accessed Sep. 14, 2023.
[60]. Linux kernel guide. Distributed Switch Architecture (DSA). Available at: https://docs.kernel.org/next/networking/dsa/dsa.html, accessed Sep. 14, 2023.
[61]. Linux kernel guide. Linux generic IRQ handling. Available at: https://www.kernel.org/doc/html/latest/core-api/genericirq.html, accessed Sep. 14, 2023.
[62]. Barham P., Dragovic B., Fraser, et al. Xen and the art of virtualization. ACM SIGOPS operating systems review, vol. 37 no. 5, 164-177, 2013.
[63]. Arpaci-Dusseau R.H., Arpaci-Dusseau A.C. Operating systems: Three easy pieces. Arpaci-Dusseau Books, LLC, 2018. Available at: https://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf, accessed Sep. 14, 2023.
[64]. Li S.W., Li X., Gu R., Nieh J. Hui J.Z. A secure and formally verified Linux KVM hypervisor. In Proc. of 2021 IEEE Symposium on Security and Privacy (SP), pp. 1782-1799, 2021.
[65]. Constantinescu C. AMD EPYC™ 7002 series–a processor with improved soft error resilience. In Proc. of 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), pp. 33-36, 2021.
[66]. Linux. Work-arounds for many known PCI hardware bugs. Available at: https://elixir.bootlin.com/linux/latest/source/drivers/pci/quirks.c, accessed Sep. 14, 2023.
[67]. Linaro. The Devicetree Specification. Current Release. Available at: https://www.devicetree.org/specifications/, accessed Sep. 14, 2023.
[68]. Linux kernel guide. Linux and the Devicetree. Available at: https://www.kernel.org/doc/html/latest/devicetree/usage-model.html, accessed Sep. 14, 2023.
[69]. Linux manual page. vdso(7). Available at: https://man7.org/linux/man-pages/man7/vdso.7.html, accessed Sep. 14, 2023.

[70]. Linux kernel guide. Scalable Vector Extension support for AArch64 Linux. Available at: https://www.kernel.org/doc/Documentation/arm64/sve.txt, accessed Sep. 14, 2023.

Информация об авторах / Information about authors

Никита Александрович СТАРОВОЙТОВ – магистрант и ассистент кафедры прикладной математики. Сфера научных интересов: кластеризация, анализ текстов.

Nikita Alexandrovich STAROVOYTOV – master student and assistant at the department of Applied Mathematics. Research interests: clusterization, text analysis.

Николай Андреевич ГОЛОВНЕВ – магистрант кафедры прикладной математики. Сфера научных интересов: анализ текстов, JVM.

Nikolay Andreevich GOLOVNEV – master student at the department of Applied Mathematics. Research interests: text analysis, JVM.

Сергей Михайлович СТАРОЛЕТОВ – кандидат физико-математических наук, доцент. Сфера научных интересов: формальная верификация, model checking, киберфизические системы, операционные системы.

Sergey Mikhailovich STAROLETOV – Candidate of Physical-Mathematical Sciences (PhD), associate professor (docent). Research interests: formal verification, model checking, cyber-physical systems, operating systems.