Sixth Conference on Software Engineering and Information Management (SEIM-2021)

Peter Trifonov, Anton Podkopaev (editors)
April 17, 2021
Sixth Conference on Software Engineering and Information Management (SEIM-2021)  
Saint Petersburg, April 17, 2021  

Peter Trifonov, Anton Podkopaev (editors)

This volume contains six selected papers originally presented at the Sixth Conference on Software Engineering and Information Management (SEIM-2021), which was held online on April 17, 2021. These papers were selected in thorough single-blind reviewing process.

ISBN 978-5-6047002-0-4

Формат 215х275. Печать цифровая.  
Тираж 40 экз. Заказ №337735.  
Отпечатано в ООО "Цифровая фабрика "Быстрый Цвет".  
197022, Санкт-Петербург, наб. реки Карповки, д.5, корп.16.  
(812) 644-40-44

Originally published online by CEUR Workshop Proceedings  
(CEUR-WS.org, ISSN 1613-0073, volume 2953)

Copyright © 2021 for the individual papers by the papers’ authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message from the Editors</td>
<td>4</td>
</tr>
<tr>
<td>SEIM-2021 Organization</td>
<td>5</td>
</tr>
<tr>
<td>Viterbi Algorithm Specialization Using Linear Algebra</td>
<td>6</td>
</tr>
<tr>
<td>Ivan Tyulyandin, Daniil Berezun, and Semyon Grigorev</td>
<td></td>
</tr>
<tr>
<td>Navitas Framework: A Novel Tool for Android Applications Energy Profiling</td>
<td>11</td>
</tr>
<tr>
<td>Vladislav Myasnikov, Alexey Shaposhnikov, Stanislav Sartasov, Egor Gordienko, Olga Aphonina, and Alan Gamaonov</td>
<td></td>
</tr>
<tr>
<td>Merging Occupancy Grid Map based on Transferable Belief Model</td>
<td>20</td>
</tr>
<tr>
<td>Maxim Dobrokhalov and Anton Filatov</td>
<td></td>
</tr>
<tr>
<td>Moment Generating Function to Probability Density Function Transform Methods</td>
<td>27</td>
</tr>
<tr>
<td>Daria Gaponenko and Aleksandr Sidnev</td>
<td></td>
</tr>
<tr>
<td>Generator of Automated Tools for Program Instrumentation</td>
<td>35</td>
</tr>
<tr>
<td>Mikhail Onischuck and Vladimir Itsykson</td>
<td></td>
</tr>
<tr>
<td>Visualizing Russian Kinship Term Possessive Sequences as Family Trees</td>
<td>41</td>
</tr>
<tr>
<td>Anna Gółub, Alyona Belova, Gleb Bondarenko, and Darya Karmaz</td>
<td></td>
</tr>
</tbody>
</table>
Message from the Editors

The Sixth Conference on Software Engineering and Information Management (SEIM-2021) opens its doors to young researchers and practitioners in different areas of computer science and software engineering, providing an opportunity to present their research, discuss state-of-the-art technology and engage in useful networking. As before, we consider SEIM to mainly focus on researchers who are just starting out their scientific careers, and hope to ease their introduction to the conference process. On the other hand, SEIM might also be of interest to more experienced researchers, who are aimed at sharing their research with a wider scientific community.

The conference welcomes submissions on a wide range of topics, including but not limited to:

- Algorithms and data structures
- Big data
- Cloud systems
- Coding theory
- Compilers
- Crowdsourcing
- Data storage and processing
- Development management
- Digital signal processing
- Distributed systems
- E-commerce / e-government
- Empirical software engineering
- High-performance computing
- Information retrieval
- Information security
- Intelligent data analysis
- Internet of Things
- Machine learning
- Mobile systems
- Modelling
- Natural language processing
- Networks and telecommunications
- (Non-)relational databases
- Operating systems
- Programming languages
- Recommendation systems
- Robotics
- Semantic web
- Social networks
- Software analysis
- Software testing
- Software verification
- Software virtualization
- Software-defined networks
- Theoretical computer science
- Visualization

This year we received 30 papers, each reviewed by at least 3 members of the Program Committee, of which six were selected for publication in CEUR-WS.org. We would like to thank the members of our Program Committee for their continuous work and contribution to the success of our conference.

These proceedings include the SEIM-2021 papers, which were selected by the Program Committee for publication in CEUR-WS.org. These papers passed not only the original review procedure, but also an additional round of post-review with the conference feedback. We thank the authors for their submissions to SEIM-2021 and hope to see them in the future.

Furthermore, we would also like to thank Tatiana Mironova, Sergey Zherevchuk and Svyatoslav Mikhailov for their great help in organizing the conference, Computer Science Center for hosting the event, and JetBrains Research for their overall support. The additional information about the SEIM conference series can be found on the conference website at:

http://2021.seim-conf.org/

Peter Trifonov, Anton Podkopaev (editors)
SEIM-2021 Organization

The conference was organized jointly with Computer Science Center and supported by JetBrains Research.

Steering Committee

Dmitry Boulytchev / St. Petersburg State University
Vladimir Itsykson / St. Petersburg Polytechnic University
Andrey Ivanov / JetBrains
Iakov Kirilenko / St. Petersburg State University
Kirill Krinkin / St. Petersburg Electrotechnical University
Boris Novikov / HSE University

Program Committee Chairs

Peter Trifonov / ITMO University
Anton Podkopaev / HSE University and JetBrains Research

Program Committee

Marat Akhin / St. Petersburg Polytechnic University
Anton Alekseev / St. Petersburg Department of V.A.Steklov Institute of Mathematics RAS
Danil Annenkov / University of Copenhagen
Daniil Berezun / JetBrains and St. Petersburg State University
Sergey Bezzateev / St. Petersburg University of Airspace Instrumentation
Nikita Bobrov / St. Petersburg State University
Natalia Bogach / St. Petersburg Polytechnic University
Dmitry Boulytchev / St. Petersburg State University
Pavel Braslavski / Ural Federal University
Timofey Bryskin / St. Petersburg State University and JetBrains Research
George Chernishev / St. Petersburg State University
Anton Filatov / St. Petersburg Electrotechnical University
Viacheslav Galaktionov / St. Petersburg State University
Natalia Grafeeva / St. Petersburg State University
Valentin Grigoriev / St. Petersburg State University
Semyon Grigorev / St. Petersburg State University
Dmitry Grigorev / St. Petersburg State University
Vladimir Itsykson / St. Petersburg Polytechnic University
Evgeny Kalishenko / St. Petersburg Electrotechnical University
Aleksandr Kapitonov / ITMO University
Iakov Kirilenko / St. Petersburg State University
Evgeniy Klyuchikov / St. Petersburg State University
Vladimir Kovalenko / JetBrains Research
Kirill Krinkin / St. Petersburg Electrotechnical University
Svetlana Lazareva / Raidix
Yuri Litvinov / St. Petersburg State University
Dmitry Luciv / St. Petersburg State University
Igor Malyshev / St. Petersburg Polytechnic University
Dmitry Mordvinov / St. Petersburg State University and JetBrains Research
Boris Novikov / HSE University
Andrey Ovchinnikov / St. Petersburg University of Aerospace Instrumentation
Oleg Pliss / Lightmatter
Nikita Povarov / JetBrains
Sergey Salishev / St. Petersburg State University
Stanislav Sartasov / St. Petersburg State University
Yuri Senichenkov / St. Petersburg Polytechnic University
Kirill Smirnov / St. Petersburg State University
Darja Solodovnikova / University of Latvia
Sergey Stupnikov / Institute of Informatics Problems RAS
Alena Suvorova / SPIIRAS
Artem Trofimov / Yandex
Anton Trunov / Zilliqa Research
Dmitry Tsetelov / Deveperts
Ekaterina Verbistkaya / St. Petersburg State University
Natalia Voloshina / ITMO University
Mark Zaslavskiy / JetBrains Research
Mikhail Zymbler / South Ural State University
Abstract—Algorithms based on linear algebra is widely used in various areas. Often programs of interest have some input data that are independent of the dataset being processed, and thus they can be optimized with respect to this input. In the paper, we study how partial evaluation affects the Viterbi algorithm as a step to research an application of partial evaluation to linear algebra-based algorithms. We evaluate the specialized multi-thread Viterbi algorithm against existing GPU-based CUDAMPF. The results show that the presented algorithm is slower but comparable to CUDAMPF.

I. INTRODUCTION

Algorithms based on linear algebra is widely used in various areas such as machine learning [1], computer vision [2], statistics [3] analysis of logic programs [4], graph theory [5], etc. One of the typical cases is querying huge database or graph processing and can be executed in days or weeks making crucial even a simple constant time optimization. One way to optimize such data processing is to use some hardware capabilities such as different kinds of parallelism or to invent a more efficient algorithm or some tricky data representation. We focus on an alternative way to program optimization based on the following observation. Often programs of interest have some input data that are independent of the dataset being processed, and thus they can be optimized with respect to this input. Partial evaluation, a.k.a. program specialization, is a well-known program transformation technique aiming to perform such an optimization [6].

In the paper, we study how partial evaluation affects the Viterbi algorithm [7] as a step to research an application of partial evaluation to linear algebra-based algorithms. First, the Viterbi algorithm is used in bioinformatics [8], speech recognition [9], and financial computations [10]. Second, it can be expressed in terms of linear algebra [11]. Third, the algorithm has two parameters: a hidden Markov model (HMM) [12] and an observations sequence. Its goal is to count a probability for the sequence to be emitted by the given HMM. Next, a sequential application of the algorithm with a fixed HMM to a big bunch of observations sequences is usual. Finally, the main part of the Viterbi algorithm heavily depends on the HMM. All the above make the algorithm a good candidate to research partial evaluation application.

The rest of the paper organized as follows. Section II describes the background. In section III the Viterbi algorithm specialization is explained. Section IV reports benchmarks results. Related work is reviewed in section V. And section VII ends up the paper.

II. BACKGROUND

In this section, we review specialization, hidden Markov models (HMM), and the Viterbi algorithm in terms of linear algebra.

A. Specialization

Specialization [6], or partial evaluation, is a well-known program transformation technique widely used when some of the input data is already known in compile time. A typical case is serial data processing when one of the input parameters is fixed while others vary. Fixed parameters are called static while other parameters are called dynamic. The idea behind specialization is that optimization of a program with respect to the static parameters together with executing the optimized program on a set of dynamic parameters may be more efficient than iterative execution of the initial program on both static and dynamic parameters.

The classical specialization example is the exponentiation function $f(x, n) = x^n$ where $n$ is static. A simple implementation is shown below.

```plaintext
function f(x, n)
    if n == 0 then 1
    elif even(x) then f (x, n/2) ^ 2
    else x * f (x, (n-1)/2) ^ 2
end function
```

All recursive calls are static, i.e., are controlled by the static parameter only, and thus can be reduced. Given fixed $n$, say 5, a typical specialized version is

```plaintext
function f_spec(x) = x * (x ^ 2) ^ 2
```

Note, sometimes specialization is useless. For example, consider the exponentiation function with fixed base $x$ but dynamic power $n$. Of course, one may use arithmetic tricks for some $x$ but in general, there is no recipe for effective specialization. Moreover, since optimal specialization is obviously
undecidable there are some heuristics to ensure specialization termination. As a result, in some cases, specialization worsen program execution. For example, it is well-known that sometimes specialization negatively affects program execution caused by code expansion [6].

B. Hidden Markov model

Hidden Markov model is a deterministic probability automaton [12]. It has the following parameters:

- \( S_{1,N} \) — \( N \) states of the automaton;
- \( O_{1,K} \) — \( K \) possible observations;
- \( B_{1,N} \) — a probabilities describing each state from \( S_{1,N} \) to be a start one;
- \( T_{1,N,1,N} \) — state transition matrix, \( Ti,j \) is a probability to go from state \( S_i \) to state \( S_j \);
- \( E_{1,N,1,K} \) — emission matrix, where \( E_{i,j} \) defines probability to emit observation \( O_j \) at state \( S_i \).

With a given observation sequence, it is possible to calculate a maximum likelihood to be at a concrete state of the HMM, i.e. to reveal hidden states, according to the sequence. HMM makes a transition between states for each observation from the given sequence.

C. Viterbi algorithm

Let’s fix the observation sequence as \( Obs \), where the length of \( Obs \) is \( lo \). The Viterbi algorithm [11] handles sequence \( Obs \) for a HMM. Its result is a maximum probability to reach each state of the HMM after handling \( Obs \).

First of all, HMM probabilities are transformed into negative logarithm. We define such probabilities as transformed probabilities \( t(p) \), where \( p \) is a some probability from the HMM definition.

\[
t(p) = \begin{cases} 
p > 0 : & -1 * \log_2(p) 
p = 0 : & +\infty 
\end{cases}
\]

(1)

e.g. probability 0.5 will be expressed as \(-1 * \log_2(0.5) = 1\). It is done to reduce the loss of precision.

The key idea is to use a special algebraic structure, named semiring \( \text{Min}_\text{plus} \). Elements of this semiring are floats. We define the addition’s semantic as a minimum between two floats. The multiplication symbol means addition for floats. Neutral elements are \(+\infty\) and 0 accordingly. This is an example of usage \( \text{Min}_\text{plus} \) semiring for matrix multiplication:

\[
\begin{pmatrix} 0 & 1 \\
+\infty & 2 \end{pmatrix} \times \begin{pmatrix} 3 \\
4 \end{pmatrix} = \begin{pmatrix} \min(0 + 3, 1 + 4) \\
\min(+\infty + 3, 2 + 4) \end{pmatrix} = \begin{pmatrix} 3 \\
6 \end{pmatrix}
\]

Expression \( t_{i,j} \) denotes transformed probability to get observation \( j \) at state \( i \), i.e. \( t(E_{i,j}) \). For each observation \( j \) we create diagonal matrix \( P(j) \) as follows with data from \( E \):

\[
P(j) = \begin{pmatrix} t_{1,j} & \cdots & +\infty \\
\vdots & \ddots & \vdots \\
+\infty & \cdots & t_{N,j} \end{pmatrix}
\]

The initial step of the Viterbi algorithm is to set up data according to the first observation from the observation sequence \( Obs \).

Symbol \( \times \) stands for matrix multiplication using \( \text{Min}_\text{plus} \) semiring. Column \( B \) defines a probability distribution for states to be a start one.

\[
Probs_1 = P(Obs_1) \times B
\]

The next step is to handle the rest of the sequence, where \( t \) changes from 2 up to \( lo \).

\[
Probs_t = P(Obs_t) \times T^\top \times Probs_{t-1}
\]

As a result, the column \( Probs_{lo} \) will contain transformed probabilities to be in a certain state of the HMM if observation sequence \( Obs \) is handled.

III. SPECIALIZED VITERBI ALGORITHM

Here we describe specialization of the Viterbi algorithm in terms of linear algebra. We fix HMM as a static parameter.

To the best of our knowledge, there is no stable partial evaluator maintaining parallel program transformation and providing expected results. In order to achieve specialization effect we’ve made ad-hoc generating extension, i.e. we provide an effective procedure to perform static data transformation and propagation together with handwritten specialized version of the Viterbi algorithm itself.

A. Theory

The goal is to embed data from the given HMM into the program and simplify expressions. These static data are \( S, O, B, T, \) and \( E \). Given a fixed HMM, for all possible observations \( o \in O \) the following matrices and matrix multiplications can be precalculated during the specialization phase according to the given in the previous section the Viterbi algorithm definition:

- \( P(o) \),
- \( P(o) \times B \), denoted latter as \( PB(o) \),
- \( P(o) \times T^\top \), denoted latter as \( PT(o) \).

We can precalculate these operations and memoize the results for further use by the specialized algorithm. The precalculation procedure pseudocode is shown in Listing 1, function \( \text{spec}_\text{Viterbi} \).

The specialized Viterbi algorithm is shown in Listing 1, function \( \text{run}_\text{Viterbi}, \) and works as follows. The initial step can be expressed as

\[
Probs_1 = PB(Obs_1).
\]

The rest of the sequence \( Obs \) is handled with multiplication

\[
Probs_t = PT(Obs_t) \times Probs_{t-1}.
\]

Comparing the specialized version with the initial one, there are fewer matrix multiplication operations in \( \text{Min}_\text{plus} \) semiring. For the first step, it is one matrix assignment instead of multiplication. For the remaining observations, we need to perform only one matrix multiplication against two. Thus, the initial Viterbi algorithm in terms of linear algebra requires to perform \( 1 + 2 \times (lo - 1) \) matrix multiplications, where \( lo \) is the \( Obs \) length, while the specialized version requires only \( lo - 1 \) multiplications but requires additional memory to keep the precalculated matrices.
Since matrix multiplication in semiring $\text{Min}_+$ is associative one can handle two observations by

$$\text{Probs}_t = PT(\text{Obs}_t) \times \text{Probs}_{t-1}$$

$$= PT(\text{Obs}_t) \times (PT(\text{Obs}_{t-1}) \times \text{Probs}_{t-2})$$

$$= (PT(\text{Obs}_t) \times PT(\text{Obs}_{t-1})) \times \text{Probs}_{t-2}$$  \hspace{1cm} (3)

Since we know all $PT(o)$, we can precalculate these multiplications, i.e. compute $K \times K$ matrices, and use them as needed. This method can be extended to handle more observations at once, e.g. for three observations:

$$\text{Probs}_t = PT(\text{Obs}_t) \times PT(\text{Obs}_{t-1}) \times PT(\text{Obs}_{t-2}) \times \text{Probs}_{t-3}$$  \hspace{1cm} (4)

For three observations there are $K \times K \times K$ evaluated matrices accordingly. We name equation 2 $1$-level specialization since only one observation handling is precalculated. By analogy, we name equations 3 and 4 by the second and the third specialization levels, and so on. $N$-level can be computed with $PT$ and $N-1$-level as follows: for all $o$ multiply $PT(o)$ for all matrices at the previous level.

B. Some implementation details

To perform matrix operations, we used SUITE-PARSE:GRAPHBLAS [5]. It is a high-performance implementation of the GRAPHBLAS [13] standard, which is intended to handle graphs, e.g. hidden Markov model. Also, it defines various linear algebra primitives, such as $\text{Min}_+$ semiring. Our implementation uses custom formats to define HMM and observation sequences simplifying data parsing. The full source code is available online [14].

IV. Evaluation

In this section, we compare the specialized Viterbi algorithm against the initial one and CUDAMPF [15].

CUDAMPF is a GPU implementation of the Viterbi algorithm. Since the Viterbi algorithm can be effectively paralleled, a GPU is a suitable choice. CUDAMPF works with hidden Markov models from bioinformatics. A model describes protein family. An observation sequence specifies a protein and contains amino acids. If a probability to be in some special state of the HMM is higher than a threshold, than protein belongs to the protein family.

We took 24 HMMs from CUDAMPF repository\(^1\). All HMMs have a different number of states but the same structure. Since these HMMs have a slightly different definition, we implement a converter into our custom format. We evaluate our solution on three different datasets. Two datasets are randomly generated, each contains three sequences consisting of 3500 and 7000 observations respectively. The third dataset is real-world 16 proteins taken from PFAM [16] database. The length of the proteins varies from 38 to 7096 observations. The number of possible observations, i.e. $K = 20$, for all datasets.

We run experiments on Ubuntu 20.04, Intel Core i7-6700 3.40 GHz, 64 Gb RAM, NVIDIA GeForce GTX 1070. Each implementation with concrete parameters was run 10 times, and a median was taken as a result. We evaluate only the first and second level specialization of the presented algorithm, since memory used for memoization grows by an exponent. For the third level the out-of-memory exception was thrown.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CUDAMPF Initial 1-level 2-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-world</td>
<td>8796 15864 12036 298209</td>
</tr>
<tr>
<td>3 x 3500</td>
<td>4854 10765 8062 215329</td>
</tr>
<tr>
<td>3 x 7000</td>
<td>9209 21062 16152 384764</td>
</tr>
</tbody>
</table>

TABLE I: Result run time (both specialization and the Viterbi algorithm), ms

The results (see Figures 1a, 1b, 1c and Table I) show that the first level specialized version of the Viterbi algorithm, as expected, is faster, than the initial one. Unexpectedly, the second level implementation is significantly slower comparing to the initial and the first level implementations, and it is

\(^1\)https://github.com/Super-Hippo/CUDAMPF (date: 2021-12-02)
not shown in some figures. One of the reasons for such a slowdown is the increased memory consumption. Nevertheless, even on a workstation with 8 CPU threads, CPU-based first level specialization outperforms GPU-based CUDAMPF on the small HMMs used in CUDAMPF benchmarks\(^2\). After all, these results are comparable with ones from CUDAMPF. It is also worth noting, that the parallel Viterbi algorithm, expressed in terms of linear algebra, is easier to implement than the CUDAMPF dynamic programming version, since the abstraction level is higher. All the above proves the specialization of the Viterbi algorithm is applicable in practice.

V. RELATED WORK

There are a lot of works, where specialization was successfully applied.

\(^2\)For evaluation we used the exact set of HMMs from CUDAMPF benchmarks.

In [17] it was shown that the result of specialization of the naïve pattern match algorithm to some fixed pattern can be behavioural equivalent to the Knuth, Morris and Pratt algorithm. This result is often used as a strength test for partial evaluators to be “good enough”. Since we consider the concrete algorithm specialization only, the test is not applicable in our case.

A partial evaluation was used in ray tracing [18] by P. H. Andersen. The author optimizes a ray tracer, gaining speedups from 1.8 to 3.0 depending on the meta-parameters and compiler. The main performance improvement was reached with constant propagation and unrolling loops. It can be done by an optimizing compiler, but this partial evaluator is aggressive. That means sometimes a specialized algorithm can have an enormous code size and lead to performance regression. The static data was directly written inside source code, while our solution can run without any files’ modifications.
A. Tyurin, D. Berezun and S. Grigorev have applied specialization to the naïve pattern match string search algorithm, implemented as a GPU program [19]. They got performance improvement up to 8 times in some cases. GPU has a lot of simple algebraic logic units. All of them need to take data to work with. It means a data cache of GPU is a bottleneck. Using specialization, static data was moved to a code cache. Such transformation makes data cache miss less possible. One may call it a "hardware specialization".

C. Sakama et al. used linear algebra as a logic programs representation [4]. The authors introduce partial evaluation as a part of the algorithm to find a logic model of a program. If specialization is used, run time is decreased by 10 times.

VI. DISCUSSION

There are some possible research directions and future work. The Viterbi algorithm specialization is the first step to find out if partial evaluation can be effectively applied to the linear algebra algorithms.

First of all, the next step is to run benchmarks at a GPU. SuiteSparse:GraphBLAS [5] is the reference CPU implementation of the GraphBLAS [13] standard. There are some GPU implementations, such as GraphBLAST [20] and GBTL [21], but to our knowledge, they are unstable.

One can try to apply partial evaluation to the other algorithms in terms of linear algebra. These experiments will reveal the limits of the specialization to such algorithms. There is a high chance that such experiments can be successfully used in production.

Since partial evaluation can lead to a performance increase, it can be useful to implement a linear algebra library with specialization primitives. It will let to develop more effective applications with linear algebra algorithms in less time.

Another approach is to do hardware partial evaluation, e.g. to make FPGA, where specialization program with static data will be embedded as a scheme.

VII. CONCLUSION

In the paper, we study an application of partial evaluation to the linear algebra-based algorithms on a particular example — the Viterbi algorithm with an HMM being fixed, i.e. static parameter. The specialized version of the Viterbi algorithm is presented. Our experiments show that on real benchmarks the presented algorithm can be comparable to the existing GPU-based Viterbi algorithm implementation CUDAmpf. Thus, the proposed approach is applicable in practice and further partial evaluation application to the linear algebra-based algorithms is a promising research direction.

REFERENCES

[14] Implementation repository. URL: https://github.com/IvanTyulyandin/Lin_alg_Viterbi (Date: 2021-13-02).
[21] Graphblas template library (gbtl), v. 3.0. URL: https://github.com/cmu-sei/gbtl (Date: 2021-12-02).
Navitas Framework: A Novel Tool for Android Applications Energy Profiling

Vladislav Myasnikov∗, Alexey Shaposhnikov† and Stanislav Sartasov‡,
Egor Gordienko§, Olga Aphonina¶ and Alan Gamaonov∥,
Saint Petersburg State University

∗ vladislav.myasnikov@bk.ru
† st069259@student.spbu.ru
‡ stanislav.sartasov@spbu.ru
§ egor.gordienko.00@gmail.com
¶ o.aphonina@gmail.com
∥ st061582@student.spbu.ru

Abstract—In a modern world smartphones became a commonly used electronic devices performing numerous day-to-day tasks and much more. But they require battery power to operate. It is well-known that computationally intensive programs as well as those using different smartphone peripherals tend to discharge the battery much quicker than their less intensive counterparts, leading to a decreased operating time. To make an application energy-aware, developers need tools to analyze its energy consumption. In this paper we present an open-source software framework to create such tools, Navitas Framework, as well as its practical application — an Android Studio IDE plugin to profile energy consumption of an application, Navitas Profiler. We describe design and architecture of the framework, outline plugin capabilities and demonstrate its usage.

Keywords—energy efficiency, green software engineering, mobile development, energy profiling, Android, software power metering

I. INTRODUCTION

“My phone discharged!” In a modern world this phrase is a known source of frustration for billions of people. By the end of 2023 it is projected for a number of smartphone users to reach 4.3 billion with Android OS still being a leading mobile OS [1].

As smartphone components require electrical power to operate, a battery provides a fixed level of voltage and a variable current. When battery charge level is low, voltage level drops beyond a certain threshold, where it is not enough for smartphone components to work properly. As smartphone peripherals tend to consume more power during high workloads than during idle states, it can be seen why computationally intensive or network-intensive software spend battery charge more rapidly than its less intensive counterparts. The less time remains for the user to work with an application, the worse the user experience.

While advances in materials and electronics in the last years helped to offset this problem by introducing more capacious batteries or power-saving processors, the issue of energetically inefficient software is still important. Green software development views energy consumption considerations as important as performance metrics [2], but it is still not so popular amongst developers [3]. One of the valuable results in this field are energy-efficient refactoring — code changes that don’t change application behavior, but reduce its energy footprint [4].

But how bad is particular code from an energy consumption perspective? Do we really need to apply a refactoring to it? To answer these questions a developer needs to measure power drain of an application, module, procedure or test, therefore a power metering tool which is reliable and easy to use is required. We think that such a tool should become an integral part of development culture instead of being used only during energy bug fixing sessions.

Some successful frameworks and tools were created previously, but we identify two essential gaps we would like to fill with our software:

- Overall level of IDE integration is still regretfully low. Only a handful of tools can be run inside IDE, and among those who can, not every profiling scenario useful to developers is supported.
- To our knowledge, no power metering software considers multi-threading to be an important factor. Multi-threaded applications, while being common under mobile operating systems for more than a decade, introduce an additional level of complexity to energy consumption measurement or estimation.

With this considerations in mind we present Navitas Framework and Navitas Profiler plugin for Android Studio IDE.

This paper is organized as follows. In Section 2 we lay out the context of our work in terms of Android OS capabilities along with the related works. In section 3 the design of Navitas Framework is discussed. Section 4 adds more details to specific modules of a framework and describes the capabilities of the Navitas Profiler plugin. Experiments used to prove framework validity are described in Section 5. In Section 6 we list current limitations of our framework and outline future work.
II. OVERVIEW

A. Related works

We conducted a systematic literature review (SLR) on the power metering frameworks for Android OS [5]. The main classification point for different approaches in software energy consumption estimation is whether power is directly measured or indirectly estimated.

Direct measurement means that some metering agent, either internal or external to Android device, is directly measuring voltage, current or even power. The simplest way is to connect external digital multimeter to battery contacts of a smartphone and work with an application in question or launch a unit test. At the same time multimeter will write power readings [6], [7], [8]. Alternatively one can use internal power sensors of a smartphone itself [9], [10], [11]. Such a tool can get a good power consumption reading for entire smartphone, experimental design may narrow it down to a level of a single application. For a more detailed report the source code is instrumented with additional logging instructions to mark method or code beginning and end, and two sources of data — power readings and application execution trace [9], [4] — are generated and aligned. Instrumentation energy overhead should also be estimated and accounted [4].

Indirect estimation means that profiling software is aggregating some code execution information and relates it to energy consumption using some model, therefore we also call this approach model-based.

One of the ways to build a model is to estimate bytecode energy consumption by evaluating an energy consumed by a number of specific instruction types using weighted sum (for example, conditional statements, loops, float-point arithmetic etc.) [12]. Direct measurement framework can be used to obtain weight coefficients. This approach allows to evaluate energy consumption of a test run without even running it. It can be further advanced by accounting library functions [4].

Another way to make an indirect estimation framework is to base an estimating model on time percentage that specific hardware component was active or worked at a specific frequency from the finite set [13]. Each value is multiplied by regression coefficient, and the total sum is found. While it is not difficult to track system events like peripheral switching on and off to aggregate this data, the challenge lies in CPU power estimation. Let’s model CPU power consumption as a sum of idle consumption and active per core consumption. For the sake of discussion let’s assume those values are constant. CPU consumes more power when multiple cores are active compared to a single-core computation. Due to the Peukert’s law [14] the real battery capacity under higher power consumption is smaller than under a lower one, and this difference is considerable [13]. Thus using a single coefficient for each instruction type is insufficient for accurate power consumption estimation in a multi-threaded case. A model to account this effect requires power drain value for each combination of peripherals currently turned on [13]. This information is challenging to be obtained experimentally, requires direct measurement framework and significantly complicates model computation.

B. Industrial grade frameworks

Only a handful of the reviewed frameworks has a source code or an application available for reuse, and even less support modern versions of Android OS. However, there is a number of tools that are used in the industry to assess application energy consumption.

BatteryHistorian [15] is a background system events analyzing utility for Android OS. It is a direct measurement framework using internal smartphone sensors through OS API, but energy consumption is represented as a battery charge percentage which is prone to battery degradation and not precise enough to estimate power drain for a short-running test.

Android Studio Energy Profiler [16] is a plugin for Android Studio IDE. It shows power drain in relative terms during application execution. While the information on the model itself is very sparse, we consider it to be indirect estimation framework based on a component working time model. We carefully note that this model is device-agnostic and it is not clear how accurate does it represent real device energy consumption.

Our conclusion is that while there were interesting research projects, and some of the tools see practical use, only Android Studio Energy Profiler is integrated with Android Studio IDE, but it has its own limitations.

C. Multi-threading in Android OS and its challenges to energy consumption measurement

Android OS fully supports multi-threading. Multi-threaded applications are common nowadays, as it is a good way to distribute computational load among the processor cores. This idea became even more lucrative with the advent of big.LITTLE2 processor architecture [17], where processor cores are no longer homogeneous, but instead they are grouped into clusters based on their computational capabilities and power requirements.

Every Android application is being executed as a separate process having its own threads. A process currently running in foreground or able to appear on the screen has its threads assigned to foreground control group, while other processes have their threads assigned to background control group. It is important to note a heavy disparity in processor time between those groups: foreground threads get about 95% of time, while background threads get the rest of it.

To our knowledge, none of the energy consumption frameworks specifically target multi-threaded applications, and only an approach from a single paper can be directly extrapolated to multi-threaded case [13], however the corresponding source code or an application is not available. It is a challenging task to understand individual thread impact on overall energy consumption. It is possible to understand when the thread was actively executing by ftrace system calls [18] and to...
homogenize peripheral energy consumption over time by controlling DVFS governor. However, as noted previously, CPU power consumption greatly varies depending on a number of cores under active payload. It is currently an open question if there is a correlation between this variable power consumption and the data in CPU power profile and, more importantly, how to adjust measurement and analysis processes to account this variation. To our knowledge, no other papers or profiling tools documentation answer these questions as well.

III. Navitas Framework DESIGN

A. Conceptual approach

Direct measurement approach can easily be extended to a multi-threaded case. If program execution trace is not used in the framework, then application energy consumption is an aggregate of energy consumption for all its threads, so the measurement process would not be different from single-threaded case. While our understanding of underlying energy consumption would be limited, in some scenarios this is exactly what is required — to measure total power footprint of an application. If execution trace is used then thread ID information can be added to each trace entry, and by splitting execution trace by thread IDs one may obtain traces for each thread in application. Aligning this information with multimeter power readings will give us a timeline of energy consumption with respect to modules, methods or functions being executed at that time.

A key component for the metering software using a direct measurement approach is a hardware sensor. Sensors capturing momentary current and voltage can be used for power sensing as well as dedicated power sensors. In general, smartphone electronics operates at a constant voltage, and a good power estimate could be achieved by using only an ammeter and a manufacturer specification for peripheral working voltage as a constant\(^1\). If such a device is integrated in a smartphone itself, we call it an internal power sensor, otherwise it is an external sensor.

Android OS provides a number of APIs for getting power reads from internal sensors. With a root access to a smartphone a developer may directly read files from `proc` folder for momentary power state [7]. Alternatively, one can use `batterystats` power logging utility and download it after test case is finished [19]. Both these options allow a high degree of customization, i.e. they may or may not include information of currently active peripherals like Wi-Fi or GPS modules.

External sensors like Monsoon [20] cost money to acquire, so their acquisition might be a limiting factor. Being an external device to a smartphone they don’t capture information on active peripherals at the time of metering. When aligning application traces with external measurement data, in order to properly synchronize time marks one should synchronize timings between different devices [7].

\(^1\) Battery voltage level drops with the discharge, but it is pretty constant for Li-Ion and Li-Pol batteries from 90% to 20%

Both types of sensors can produce a series of time marks and instant power readings. The main deciding factor is the frequency of analog-to-digital converters. For external sensors a typical value is 10 KHz, therefore power difference at intervals of 100 µs can be captured. This frequency is considered in the literature as appropriate for capturing code power consumption on a method level [12], and in our observations the most power-consuming methods typically run at least for a 1 ms or in loops lasting longer than that. In the case of internal power sensing frequencies of 1 Hz or less are not unheard of [21], and that severely limits a number of profiling scenarios.

For the model-based approaches, time-based models are already well-suited for multi-threaded scenario, however special calibration should be done to address simultaneous multi-core processor and graphic card load, as their energy consumption might change non-linearly when a number of active cores or GPU multiprocessors increases. As for the models based on instruction or function energy consumption estimation, this approach is limited in multi-threaded environment. Power coefficients for specific instructions may not reflect underlying peripherals usage. In real life scenarios programs are interacting with smartphone peripherals which are at least as consuming as CPU [22] and are independent from processor execution. While a function call to decrease the screen brightness might be estimated in terms of power drain, the side effect of reducing screen power footprint cannot be reduced to processor instruction energy consumption. Second, library functions power consumption is often dependent on its input length (i.e. data transmission functions) and cannot be accurately presented by linear coefficient.

The following points were the most important when we made a decision on the conceptual approach for the Navitas Framework:

- External sensors require money to acquire and a certain qualification to be properly used.
- While direct measurement approach using internal sensors was promising at first, we quickly found out that on our test devices sensors’ frequency was not stable and high enough for practical use.
- Instruction and function-level power estimation have inherent conceptual flaws for multi-threaded analysis.
- Time-based models often have a support from manufacturers in the form of `power_profile.xml` — an XML file containing the values of electrical current for various modes of operation of the devices installed in the smartphone. This file can be obtained without root access and can be used as an initial set of weighted coefficients for our model.
- Execution data for time-based models can be obtained using Android API calls and utilities.

In the end, we choose Navitas Framework to follow time-based indirect estimation approach.

B. Reporting units

From our SLR we knew that there is no uniformity in reporting units that other frameworks use [5]. Some frameworks
used absolute values like Joules, Watts or Amperes, other used battery discharge percentage or relative non-dimensional units [16]. As raw data from power_profile.xml contained values measured in milliamperes (mA), voltage can be found in device specification, and execution time is obtained from test or application execution trace, Joules and battery charge percentage were two main reporting units candidates.

Each recharge cycle for Li-ion batteries inevitably shortens their discharge time under the same drain, therefore estimating battery time change can only be done with a context of how old a battery is and how many recharge cycles it was run through [14]. As controlled experiments tremendously differ from real-time smartphone usage where screen brightness is high and multiple peripherals may be active at the same time, high battery drain might be observed. Because of Peukert’s law resulting battery time from such a drain can be estimated, but it will be lower than in experimental environment. We conclude that reporting in Joules results in a simpler estimation model.

C. Instrumentation and its overhead

Source code instrumentation is not necessary if a it can be manually updated with all necessary logging information, however automated approach is preferred, for instance by applying source code transformation using javassist library [23] or BCEL [24]. We concur with the literature that method start and end should both be instrumented to provide a proper call hierarchy [9]. Instrumentation trace from a code execution should also be profiled for energy consumption in order to deduct it from total power spending for better metering accuracy [4]. For multi-threaded case instrumentation code should be adjusted to include a thread ID for each log entry. We decided to calculate contribution of each device at the measurements results analysis phase.

D. Tests vs. application runs

There are two ways of deploying an instrumented application for power profiling ([25], [7], [6]). Tests are designed for conducting repeatable executions of the same scenario under the same conditions, and they are more suitable for controlled experiments. Different test runners like MonkeyRunner, JUnit or Espresso can be used for Android applications ([7], [12]). Application run means launching the entire instrumented application on a test device and conducting some manual interaction with it. In general it can’t be as repeatable as a test due to human involvement, but it helps to make an overview of general application power drain. A the same time tests may be checking application in an uncharacteristic way compared to a real usage.

Overall, in our opinion the choice between application runs and tests lies in a particular profiling scenario, because the boundary between the use cases for both approaches is blurry. While Navitas Framework is agnostic to the level of workload execution, we choose test runs due to simplicity of integration.

IV. Navitas Framework AND Navitas Profiler

IMPLEMENTATION

A. High-level overview

Android Studio is currently one of the most popular IDEs among Android developers [26], and while at the time of writing it is a natural choice to develop a profiling tool for it, a more general approach would be to develop a framework based on the current compiling tools and practices and a specific extension or plugin for Android Studio.

Internally, Android Studio runs a set of Gradle tasks to compile and run Android applications. Those tasks are IDE-agnostic and can be reused in other IDEs. We extend this set of tasks with Navitas profiling tasks, and the overall workflow can be seen in Fig. 1.

Our first custom task instruments the application code at the compilation stage with additional instructions to log execution trace and thread information. After the compilation is completed, we additionally initialize target device logging facilities to trace component activity information. Then after the test run is finished we download these logs along with execution traces to a computer and format component usage statistics report.

Various test runners can be supported independently from power profiling Gradle tasks. Currently we use JUnit as a default test runner, and its capabilities are fully supported. Both single tests and test sets can be profiled, and the profiling results are shown for each test (see Fig. 3). Proof of concepts were also made for Espresso and MonkeyRunner frameworks.

Navitas Profiler plugin for Android Studio is an orchestrator. It manages what code to instrument, what tests to run and what device to use for it. It also handles the energy coefficients for target devices and it also runs the model against obtained component usage statistics and displays energy consumption report to the user.

B. Framework instrumentation

The instrumentation of application code occurs during compiling and packaging. We used Transform API library which allows third-party plugins to manipulate compiled files before they are turned into dex files.

To transform the code one needs to create a class and implement Transform interface to register the code transformation. The main instrumentation logic is implemented
in the overridden `transform` method. For each of the instrumented files, the necessary imports are added to the beginning. Furthermore, for all methods of each class, methods of the `Javassist` library are applied — `insertBefore` and `insertAfter`. A string containing Java code is passed as a parameter. Even though Kotlin is now the primary development language for Android, many applications are still written in Java, so adding Java code allows us to ensure the correct operation of the original programs in both languages, as full interoperability is maintained between Kotlin and Java. Our experience shows that code instrumentation at compilation phase is faster than APK disassembling and instrumenting mentioned in earlier works.

Logcat is a command line tool that prints a log of system messages, including a stack trace, when the device generates an error, and other messages may be written by a custom application using the `Log` class. Logcat is convenient to use because in addition to the message itself it adds information about the current time (up to milliseconds), process ID and the thread ID. The code we instrument into methods adds method name to execution trace logs and accesses system API to trace component activity. Activity data is then passed to Logcat.

C. Framework component activity data acquisition and model formulas

Among the components of a smartphone which affect energy consumption the most important is the processor. Android OS has special `time-in-state` files in the `/sys` directory that contain information about the time each processor core spent at a specific frequency. The data in these files is stored in pairs like `<frequency><time>"`. There are exactly as many of these pairs as there are different frequencies a particular processor core supports. Time is measured in 10 milliseconds units, and it is counted from the moment the corresponding driver was installed to measure processor data.

Our model for the processor energy consumption is therefore straightforward: for each core and for each frequency it operated, one needs to take the time difference between method ending and beginning at the end of the method and at its beginning. Thus we obtain the time method worked at each of the frequencies. Then by multiplying those timings to corresponding weight coefficients in a model and summing the result we obtain the total energy spent by the processor to execute the measured method. Currently, the model weight coefficients are based on the data in `power_profile.xml`. To get the coefficients we multiply the values stored there in milliamperes (mA) to the nominal CPU voltage. In the end the final formula for the CPU power consumption is the following:

\[ E_{CPU} = \sum_{i \in \text{cores}} \sum_{j \in \text{freqs}} (\text{endTime}_{ij} - \text{startTime}_{ij}) \times C_{ij} \]

where `startTime_{ij}` and `endTime_{ij}` — the time the i-th kernel stays at the j-th frequency at the time of entering and exiting the method, respectively, \( C_{ij} \) — coefficient from our model.

Similarly to CPU, screen power consumption can be calculated by multiplying the time screen was turned on at different brightness levels to model weight coefficients and summing the results. These coefficients are obtained by interpolating power values from the `power_profile.xml` file for minimum and maximum brightness. Timing information is obtained from `batterystats` log, which contains screen brightness change events. Each event includes time since last `batterystats` reset in milliseconds and one of the five brightness levels: dark, dim, medium, light, bright.

D. Framework execution trace and component usage statistics acquisition

After all the tests are completed, execution logs and component status are downloaded from the test device. As logcat logs contain a lot of additional information, but each entry type structure is regular, pattern matching using regular expressions is used to extract actual data. Then thread execution traces are separated from one another, and the obtained data is aggregated into a JSON files. These files are then processed by high-level tools based on `Navitas Framework`, but we also keep the file format open if a user wants to process the data with some custom tool.

E. Navitas Profiler implementation

Navitas Profiler is a plugin for Android Studio based on `Navitas Framework` tasks and developed to provide a practitioner a high-level tool to profile energy consumption of Android applications during the development process and visualize power consumption based on profiling results while being integrated with the IDE itself.

The plugin adheres to a three-layer architecture (Fig. 2):

- **domain** — business logic, energy modelling utilities and energy profile management tools;
- **data** — an abstraction layer for `Navitas Framework` data acquisition;
- **presentation** — user interface classes.

To integrate `Navitas Profiler` to Android Studio project, its `build.gradle` file should be modified accordingly.

A common use case for `Navitas Profiler` is the following. User selects an Android module and its tests to run for profiling, as well as device energy profile — our weight coefficients for a particular test device. Then profiling is initiated on a real hardware device, and upon its completion.
The analyzer module receives report file in JSON format. The energy data is calculated for each method of each thread in the execution trace using the formulas discussed in Section 4.C. Note that Android API calls and third-party libraries are not profiled as no instrumentation is done for them. Then in a profiler window user can select the desired test and get detailed information about the energy consumption of all methods of the test (Fig.3).

A user can get energy consumption data for a specific method of a specific thread by selecting it in the tree view under the chart. The power readings during this method execution will be shown. Note that nested invocations will also be shown as a tree (Fig.4). Filtering execution trace by a specific thread ID is also available.

V. Experiments

We first confirmed that Navitas Framework is working as intended by accident. While studying energy impact of Android compiler optimizations under parallel line of work, we have found that the same code running in a separate thread using a separate class inherited from Runnable interface consumes a different amount of energy compared to the case where an anonymous class was used (see listings 1 and 2).

```java
class TaskClass : Runnable {
    @Override
    public void run() {...
}
...
new Thread(new TaskClass()).start();
...
```

Listing 1. Runnable as a separate class

```java
...
new Thread(new Runnable() {
    public void run() {...
}).start();
...
```

Listing 2. Runnable as an anonymous class

A test was created which ran some constant payload either under Runnable as a separate class, or as an anonymous class for 100 000 times. Test device was prepared according to guidelines we outlined in SLR [5]. In particular, Wi-Fi, Bluetooth and GPS were turned off, all unnecessary applications were removed or stopped, smartphone was put steady on a table and was not moved until all the test runs were over, there was a pause of 1 minute for a CPU to cool down, and CPU DVFS governor was set to performance for a constant maximum frequency. Thread affinity was not set for application threads. Navitas Profiler was utilized to measure energy consumption in both cases, and the result is shown in table I. On average, anonymous class case consumes less energy, however due to OS interference and thread scheduling is possible some particular test runs for the first payload are lighter than for the second one. It was confirmed by disassembling the APK that in the first case additional instructions were generated. We conclude that multiple test runs and results averaging should be incorporated into every experimental methodology concerning energy consumption.

In a second experiment Runnable creation was uniform, but the payload was different. In one case pseudo-random number generator was used with the constant seed to generate an array of $10^6$ integer numbers which was then saved to internal storage. In another case array sorting was done before it was saved. Therefore second payload would consume more energy than the first one. While the experimental design deliberately favours the first payload energy-wise, such energy issues may happen in real applications when the data is sorted when it’s not required do so. Table II, Fig.5 and Fig.6 show that Navitas Profiler is able to detect these discrepancies in both in total and individual thread energy consumption.

VI. Conclusions and future work

Both Navitas Profiler and Navitas Framework source codes are available at https://github.com/Stanislav-Sartasov/Navitas-Framework. We would like to address the following issues in our future work:
Currently the list of supported peripherals is very short, and we're working on expanding it with Wi-Fi, Bluetooth and GPS being the main priority.

While currently every method of user code is instrumented, sometimes such level of instrumentation is excessive. Sometimes only the energy consumed by the entire test is required, so instrumentation might be more lightweight. An adjustment model should also be introduced for instrumented instructions to offset their own energy consumption.

Running the same test multiple times automatically and averaging the results is a natural extension to the current testing process.

Execution trace needs to properly support exception handling as method beginning might not have a corresponding method ending in a logcat output.

Multi-threading energy consumption analysis is currently done at a pretty basic level. As we have outlined, understanding what is the energy consumption of a specific thread in an interleaved execution is a non-trivial task (see Section II.C), so adding ftrace execution data to our model as well as accounting for varying CPU power consumption under varying payload is a challenging research project.

We use the data in power_profile.xml as a basis for our model, but actual device might be inconsistent with it [6]. Currently our model is as precise as those values, so implementing a model calibration algorithm is an important next step. Testing the quality of our model against direct measurements is another mandatory direction.

However even in its current state our tool estimates peripheral energy consumption instead of battery drain and uses Joules instead of relative units, therefore we consider it to provide a valuable and concrete information on energy consumption compared to analogues. We admit that current design supports test runs and doesn’t provide real-time consumption graph like Android Studio Energy Profiler, but in our view these are just two different and equally valuable ways to work with profiling applications.

Both Navitas Profiler and Navitas Framework are used in a number of research projects of Software Engineering department of Saint Petersburg State University. As they provide a practitioner a complete set of profiling tools, we consider them to be a valuable contribution to energy profiling tools on Android platform.

REFERENCES


Fig. 4. *Navitas Profiler* detailed energy consumption of methods

Fig. 5. *Navitas Profiler* total results for second experiment

Fig. 6. *Navitas Profiler* details results for second experiment


Donghua Shin, Kitae Kim, Naehyuck Chang, Woojo Lee, Yanzhi Wang, Qing Xie, and M. Pedram, “Online estimation of the remaining energy capacity in mobile systems considering system-wide power consumption and battery characteristics,” in 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), Jan 2013, pp. 59–64.


19
Merging occupancy grid map based on Transferable Belief Model

Dobrokhvalov Maxim
Saint-Petersburg Electrotechnical University “LETI”
Saint-Petersburg, Russia
Email: modobrokhvalov@gmail.com

Anton Filatov
Saint-Petersburg Electrotechnical University “LETI”
Saint-Petersburg, Russia
Email: ant.filatov@gmail.com

Abstract—This paper presents a new algorithm for merging occupancy grid maps, the cells of which are based on the Transferable Belief Model. The developed algorithm excludes the collision of the merged maps. There are no disjoint areas in the source maps in the final map. In addition, this article explores possible modifications of the ORB algorithm descriptor using the Transferable Belief Model. Testing was conducted using the MIT Stata center dataset. The source code is available on https://github.com/Nightbot1448/TBMMapMerging.

Index Terms—occupancy grid map, map merging, Transferable Belief Model, descriptor of keypoint

I. INTRODUCTION

Occupancy grid maps are one of the variants of map representation formats that are built by robots in the process of performing the task of simultaneous localization and mapping[1]. The use of such maps is advisable when there is no a priori information about the environment. Such maps are used by autopilots, rescue robots and other robots that use information about the space around them to move. Building a map using data from only one robot is slow. Also there are a lot of errors that are difficult to fix. These disadvantages can be overcome by using more than one robot, and the maps they build can be combined into one. The difficulty lies in correcting conflicts that arise when merging maps.

Occupancy maps often use probabilistic theory to provide information about the occupancy of a cell [2]. In this paper, it is proposed to consider a different approach to storing information about space using occupancy maps. The mathematical basis for this approach is the Dempster–Schafer theory (DST) [3]. One of the directions of development of this theory is the Transferable Belief Model(TBM) [4].

Section II will describe the difference between classical probability theory and TBM. It will also be said about the representation of the cell in this work. The section III will provide a brief overview of some of the existing solutions. Section IV will describe the presentation of the map, the idea of the algorithm, and also talk about the modification of descriptors and the way of aligning maps. Subsection V-A describes the selection of a merge rule, subsection V-B compares various descriptor modifications, and subsection V-C describes the accuracy of the merge and the computation time of the algorithm.

II. MATHEMATICAL BASE

Consider some of the basis of Dempster-Schafer theory and the Transferable Belief Model. These theories are the basis of this work. For ease of further understanding, everything will be built by analogy with the states of the map cell.

\[ X = \{ h, \overline{h} \} \] (1)

\[ p(h) + p(\overline{h}) = 1 \] (2)

\[ 2^X = \{ \emptyset, \{h\}, \{\overline{h}\}, X \} \] (3)

\[ \sum_{A \in 2^X} m(A) = 1 \] (4)

Let \( h \) be some event (eq. (1)). In classical probabilistic theory, the sum of the probabilities of an event and complementary event is equal to 1 (eq. (2)). In TBM there is a transition to the superset (eq. (3)). Accordingly, the state is described by 4 masses, which show the measure of the fact that the cell is in this state. The sum of the masses must be equal to 1 (eq. (4)). Lack of knowledge or uncertainty between the states \( h \) and \( \overline{h} \) is explicitly described by the mass of \( X \), and if two sources provide conflicting information, then the mass of conflict \( \emptyset \) increases.

Consider the projection of this theory onto the cell representation of the occupancy map. The set of states is two atoms [5] \( X = \{ \text{Occupied}, \text{Free} \} \). Thus, the superset is the set of four elements \[ 2^X = \{ \emptyset, \{\text{Occupied}\}, \{\text{Free}\}, \{\text{Occupied}, \text{Free}\} \} = \{\text{Conflict, Occupied, Free, Unknown}\} \]. In this way \( m(\emptyset) = m(\text{Conflict}) \) – the mass that the cell is in a conflict state, \( m(\text{Occupied}) \) – the mass of the fact that the cell is occupied, \( m(\text{Free}) \) – the mass that the cell is free, \( m(\{\text{Occupied}, \text{Free}\}) = m(\text{Unknown}) = m(X) \) – remaining unallocated mass that reports a lack of information.

In this paper, the differences between DST and TBM are limited to the fact that TBM allows for the possibility that the conflict mass will be non-zero. This is prohibited in DST. DST always requires conflict normalization. Conflict normalization is the distribution of the mass of the conflict between all other possible states in proportion to their masses. However, in the implementation, the conflict is always normalized.
III. RELATED WORK

At the moment there are many different algorithms for merging occupancy maps. Some of the existing algorithms will be surveyed below. As sources of information on the approaches to merging maps, we selected articles describing the merging of occupancy maps, differing in merging approaches for the possibility of comparison. The articles were published in 2005-2020.

Andreas Birk and Stefano Carpin in work [6] propose an algorithm based on the Adaptive random walk algorithm with heuristics. The heuristic used in the algorithm is based on a special image similarity function, calculated in linear time. To determine the quality of the merge, a special function is introduced that shows the quality of the merge of the maps:

$$ai(m_1, m_2) = 1 - \frac{agr(m_1, m_2)}{agr(m_1, m_2) + dis(m_1, m_2)},$$

where $agr(m_1, m_2)$ - measure of map consistency, $dis(m_1, m_2)$ - measure of inconsistency. In this work, a cell can be in one of three states - occupied, free, no information. If the cell values do not match when merged, then the cell is marked as unknown. The runtime is 170 seconds on a Pentium IV 2.2 GHz processor. The main disadvantage in this work is the discrete representation of the cell state. This is a disadvantage, since it does not express a measure of confidence in the correctness of the data. Also the disadvantage is the duration of the work. However, the merge method is an advantage, since the lack of information is of higher priority than the presence of unreliable information.

In [7] Li Hao and his colleagues describe an objective function based on the probability of being occupied in the map fusion section. Also presented are some methods that are developed on the basis of a genetic algorithm to optimize the objective function. Based on this method, a general solution for the association of several robots is the additionally described strategy for indirect estimation of the relative position of robots. The proposed method can perform the task of merging, even with a larger initial map alignment error and high inconsistency inherent in the map. The merging process is averaging the values of the corresponding cells in the original maps. The operating time is 15 seconds on a processor with a frequency of 3.0 GHz. In this work, the disadvantage is the averaging of the values of the cells of the merged maps. This is a disadvantage as the conflict remains. Among the considered analogs, this algorithm has the highest performance.

Sajad Saeedi and colleagues in their work [8] extend the Generalized Voronoi Diagram (GVD) to encapsulate probabilistic information encoded in the occupancy grid map. A new construct, called Probabilistic GVD (PGVD), works directly with occupancy grids and is used to determine the relative transformation (offset and rotation) between maps and combine them. The merging process consists of several stages. Initially, there is a relative transformation between the two maps. Then the probabilities are combined and then filtered to get the final map. The data obtained as a result of the map transformation is included using the additive property of the logarithmic representation of occupancy. An entropy filter is applied to the merged map. The entropy filter compares the merged and original maps and rejects updates that increase entropy. Mutual information is defined as the decrease in entropy in cell (i, j) between the original map and the merged map. The resulting map is defined as follows: if the value of the mutual information for the cell is non-negative, then data is taken from the merged map, otherwise - from the original one. The runtime is 34 seconds on a Core2Duo 2.66 GHz processor. In this work, the main disadvantage is the operating time. The main advantage of this work is the way of resolving the conflict when merging maps.

In paper [9] by Guillaume Trehard and colleagues, the merging algorithm is based on a Transferable Belief Model for occupancy maps. The cell representation used in this paper is described in section II. The merge is performed according to the conjunctive rule [10]. When determining map consistency, the disjunctive orthogonal operator [9] is used. The main advantage of this work is the cell presentation. Using TBM for representation broadens the view of the space. As will be shown later, this work is the foundation for the current.

Andrew Howard suggests the approach [11], which uses the Bayesian inference [12] and the particle filter [13]. The algorithm works in conjunction with the task of simultaneous localization and mapping. Merging takes into account the time-reversed sequence of maps states. The resulting cell value is calculated by calculating the average of the merged maps. The advantage is the use of a particle filter as this allows to choose the best option among the many. However, averaging the merge value is a disadvantage.

The articles do not provide a link to the code, as a result of which there is no way to test it under equal conditions - on the same processor and the same dataset. As a result, the numbers are taken from the works themselves, and information about the processors and/or their frequency is indicated for comparison.

IV. IMPLEMENTATION

This section describes the map representation used in the proposed occupancy grid map merging algorithm. The occupancy map is a two-dimensional grid of cells. In this paper, the cell is based on TBM. Thanks to this, it is possible to store in the cell more information about the space. The algorithm is based on images of different occupancy map representations. To select an algorithm for detecting keypoints, a comparison of various algorithms for detecting features has been performed.

A. Map representation

The TBM cell is the main feature of the occupancy grid representation considered in this algorithm. The cell of the map is similar to that considered in the [9] algorithm.

The cell representation used in this paper is described in section II. In fact, the map can be represented as 3 different maps, each of which carry some specific information about the area on the map. It is possible to get maps showing an
occupied area, a free area, an area about which there is no information. There is possible convert a cell of type TBM to the usual probabilistic value \[4\] using the formula

\[P_{Bet}(x) = \sum_{A \subseteq X} \frac{m(A)}{|A|},\]

where \(x \in X\) are atoms, \(|A|\) is the number of \(x\) atoms that appear in \(A\). \(X\) is the set of states.

Fig. 1a provides the probabilistic representation of the occupancy map, in which an unoccupied cell corresponds to 1.0 (white), an occupied cell - 0.0 (black), a cell about which there is no information - 0.5. Different representations of the occupancy map are illustrated in fig. 1b - 1d. In these figures, white color corresponds to the fact that the entire mass of the cell is concentrated on this belief, black color - the mass of this belief is equal to 0 in this cell. As it was said, the conflict is normalized, so the mass is distributed among the three indicated beliefs. The sum of the masses of these three beliefs is 1, that is, \(m(Occupied) + m(Free) + m(Unknown) = 1\).

**B. Algorithm idea**

On the probabilistic map (1a), keypoints are extracted using the ORB [14] feature detector. Then the descriptors of the feature points in the images corresponding to the three maps listed above are calculated. The resulting descriptors are concatenated. Thus, descriptors of keypoints of the map are obtained. Next, the descriptors are matched. The matched keypoints are split according to the maps to which they belong. Then the computation of the affine transformation is performed. Once a transform is found, it is applied to the map for which it was calculated. The maps are merged using the disjunctive rule, which is described below. Alg. 1 demonstrates pseudocode of the algorithm.

**C. Comparison of descriptors**

Initially, ORB, BRIEF [15], SURF [16], SIFT [17] were considered as variants of the used detectors and descriptors.

**Algorithm 1** TBM map merging

```plaintext
imgs ← get_imgs_from_map1(keypoints1, keypoints2, descriptors1, descriptors2)
d1, d2 ← get_transform(imgs)
desc_map1, desc_map2 ← concatenate(d1, d2)
mapped ← match_and_filter(desc_map1, desc_map2)
transform ← compute_transform(mapped)
transformed ← apply_transform(map2, transform)
return merge_disjunctive(map1, transformed)
```

**Table I** Comparison of different detectors and descriptors on the maps images

<table>
<thead>
<tr>
<th>Experiment</th>
<th>ORB(1)</th>
<th>BRISK(2)</th>
<th>SIFT(3)</th>
<th>SURF(4)</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.958</td>
<td>0.889</td>
<td>0.25</td>
<td>0.0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.951</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>3</td>
<td>0.714</td>
<td>0.5</td>
<td>0.0</td>
<td>0.0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.993</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>5</td>
<td>0.96</td>
<td>0.818</td>
<td>0.0</td>
<td>0.0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0.999</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>2, 3, 4</td>
</tr>
<tr>
<td>7</td>
<td>0.999</td>
<td>0.998</td>
<td>1.0</td>
<td>0.996</td>
<td>3</td>
</tr>
</tbody>
</table>

For comparative analysis, several maps of the 2nd floor of the MIT Stata Center were built. After that, images of the probabilistic representations of these maps were obtained. On these images, about 1000 features were detected using various detectors. Next, the descriptors were computed. The imprecise count of the detecting keypoints is due to the fact that in OpenCV the BRISK and SURF algorithms do not provide the ability to explicitly specify the number of points. This count can only be influenced through some of the input parameters.

The table I shows the ratios of correctly matched feature points to the total number of matched feature points. Each line corresponds to a pair of different maps to merge. Repeated running with averaging of the results were not carried out, because algorithms for detecting and calculating descriptors do not depend on random variables.

The columns with the names of the algorithms indicate the coefficients that show the ratio of correctly matched pairs to the total number of matched pairs of features. As you can see from the table, the ratio of points that the ORB detects is better, or close to the best, so this feature detector was chosen.

**D. Maps alignment**

After calculating the descriptors, the found points must be matched. For this, brute force matching is used using the Hamming norm [18]. This method is the main one for matching binary keypoints descriptors. The Lowe test [17] is then used to partially eliminate false matches. After filtering, the matched pairs that passed the test are copied into two vectors of features that correspond to their map. The optimal affine transformation is calculated between these vectors. The result of the function operation is the \(2 \times 3\) matrix \([R|t]\), where \(R\) is the rotation matrix \(2 \times 2\), \(t\) is the translation vector \(2 \times 1\).
The found transformation is applied to the map for which it was calculated.

E. Merging using the disjunction rule

The conjunctive rule is used if all sources are considered reliable, disjunctive – if at least one source is considered unreliable. Maps, which are built by robots, can have various anomalies, such as incorrect scales (a robot in some area covered more/less distance than shown on the map), inconsistency of angles on the map and in reality (rotation error), and others. Therefore, it was decided to use the disjunctive rule, which allows you to exclude areas that have strong differences from the resulting map. For a comparison of conjunctive and disjunctive rules, see V-A.

Conjunctive rule:

\[
m_{1\cap/2}(A) = \sum_{B\cap/C=A} m_1(B) m_2(C) \quad (5)
\]

Disjunctive rule:

\[
m_{1\cup/2}(A) = \sum_{B\cup/C=A} m_1(B) m_2(C), \quad (6)
\]

where \( A, B, C \in 2^X \neq \emptyset \).

After the maps are aligned, they are merged using the disjunctive rule (6). Since the maps are aligned, an iterative traversal of the cells occurs. For each pair, the specified rule is applied, and the information obtained during the merging is set in a cell with the same index as those taken in the original maps.

V. Experiments

The algorithm for searching for transformation and merging of maps was described above. It has been said that there are different merge rules in TBM. As part of the work, conjunctive and disjunctive rules are considered. The reasons for choosing a disjunctive merge rule will be presented below.

In the section IV-B it was said that descriptor concatenation is used. This section will look at the various modifications to existing descriptors and compare these modifications. The accuracy of the found transformation is determined by finding the distance between the coordinates of the keypoints of the first map and the coordinates of the keypoints of the second map after applying the found transformation. This will be discussed below.

A. Comparison of conjunctive and disjunctive rules

As mentioned above, the conjunctive rule is used when all sources are considered reliable, and the disjunctive one is used when at least one source is not. The difference between merges using conjunctive and disjunctive rules is considered in the following example. The original map is shown in fig. 2a. Fig. 2b - 2c are represented by 51% of the left and right parts of the original map with the same size of the original map, the remaining 49% are set as cells, about which there is no information. Fig. 2d is demonstrated the result of merging using the conjunction rule, so the resulting map contains the parts that are present on both merged maps. The cells that are in the overlapping area have reduced the mass of the belief corresponding to the lack of information about this cell and redistributed it according to the rule (5). This can be seen...
If attention is paid to the overlapping 2% vertically of the resulting map – the color has become whiter, that is, the mass of the belief that the cell is free, and therefore the likelihood that this cell is free, has increased. Fig. 2e is shown the result of merging using the disjunction rule. So only that part of the original map that is present on both merged maps gets into the resulting map. The choice of the disjunctive rule is explained by the fact that the absence of information is of higher priority than the presence of unreliable information.

**B. Comparison of descriptor modifications**

Comparative analysis of different modifications of the 128-bit ORB descriptor can be seen in the table II. The columns contain the following information: 1 – the number of keypoints to be detected initially; 2 – Lowe coefficient used when filtering false matches; 3 – the ratio of the number of correctly matched features to the total number of matched features \( \xi \) (experiments were carried out on 10 different pairs of maps); 4 – the columns showing the highest coefficients indicated in columns 3. Modifications in column 3: a – approach without modification of descriptors; b – descriptors concatenation: for each keypoint, three 128-bit descriptors are concatenated, resulting in a 384-bit descriptor; c – using the logical and operation: applying a bitwise logical and for three 128-bit descriptors, the result is a 128-bit descriptor; d – applying a logical or operation: applying a bitwise logical or for three 128-bit descriptors, the result is a 128-bit descriptor; e – applying a logical exclusive or operation: applying a bitwise logical exclusive or for three 128-bit descriptors, resulting in a 128-bit descriptor.

It can be concluded that the concatenation of feature point descriptors obtained using TBM almost always shows a greater value of the \( \xi \) ratio than any other proposed modification. In most cases the value of the ratio \( \xi \) when using concatenation is not less than when matching the descriptors of keypoints calculated for the probabilistic representation of occupancy maps.

The dependence of the ratio \( \xi \) on the number of initially extracted features (at a fixed value of the Lowe coefficient) can be seen in fig. 3a. Based on this graph, statistically significant differences were seen with a confidence level of 0.95 when extracting 100, 1000 and 2500 keypoints. In the implementation of the algorithm, the default value is set to 1000. The dependence of the ratio \( \xi \) on the value of the Lowe coefficient (with a fixed number of initially extracted keypoints) can be seen in fig. 3b. Based on this graph, statistically significant differences with a confidence level of 0.95 were seen with a coefficient value of 0.7 or 0.8. In the implementation of the algorithm, the default value is 0.7.

**C. Accuracy of the transformation**

Viny_SLAM [19] allows building occupancy grid maps based on TBM. Compared to analogs such as FastSLAM [20] or Credibilist SLAM [9], Viny_SLAM is verified on the MIT Stata Center dataset. Using the specified SLAM algorithm, various maps of the 2nd floor of the MIT Stata Center were obtained. After that, the algorithm proposed in this work was applied to pairs of maps. During the operation of the algorithm, the keypoints of the maps were extracted and matched. Then the calculated transformation was applied to one of the sets of keypoints. For pairs of matched points, the distances between
these points were calculated. The table III shows the minimum, maximum and geometric mean distances in meters between the matched feature points after applying the transformation. Repeated experiments gave the same values. Based on the values of the geometric mean in the table, it can be concluded that the matching error does not exceed 0.35 meters. This error is due to the fact that not all keypoints are matched correctly.

Fig. 4 shows the maps to be merged and the result of the algorithm. Maps are shown in probabilistic representation. Map cell size is $0.1 \times 0.1 \text{m}^2$. One pixel on the image of the map used to construct descriptors corresponds to one cell, thus one pixel corresponds to $0.01\text{m}^2$.

Table IV lists the areas of the maps resulting from this experiment, as well as the average fusion time and standard deviation for 10 experiments for each pair of maps. The processing time was calculated on an Intel Core i5-8300H 2.3 GHz processor. This operating time is much longer than that required for real-time operation, but the algorithm was not originally intended for such use. Within the framework of the considered analogs, the proposed algorithm shows a faster operating time. One of the possible applications can be the construction of models of buildings in which people are dangerous. This can be used to evaluate the structure for subsequent deconstruction.
VI. CONCLUSION

In this paper, a TBM cell representation of an occupancy grid map has been described. This cell allows you to store more information about the space than a probabilistic representation. It also was compared different detectors of keypoints on the map image in order to choose the one that would be used in the algorithm. As a result, ORB was chosen. The reason is that the ratio of correctly matched pairs to the total number of matched pairs of features was higher or close to the highest. After that, various possible modifications of descriptors were considered to improve the accuracy of matching of keypoints. As a result, it was concluded that concatenation gives better results than any other modification, and also in most cases no worse than the original descriptor. Both disjunctive and conjunctive rules in TBM could be used to merge maps. However, preference was given to the disjunctive rule, since the maps may contain various errors, and this rule allows you to cut off inconsistent areas. At the end, the results of the algorithm were presented, as well as an assessment of the accuracy of the transformation search.

As a further development of this work, the clustering of keypoints and the verification of the existence of a transformation between clusters of two maps is considered. If a transformation is found, then it is possible to apply the transformation to an area of the map containing a cluster of special points. After applying a transformation to a part of the map, it is possible to merge using one of the merge rules in the Transferable Belief Model.

REFERENCES


Moment Generating Function to Probability Density Function Transform Methods

1st Darya Gaponenko
Peter the Great St.Petersburg Polytechnic University
St.Petersburg, Russia
daria.gaponenko@yandex.ru

2nd Aleksandr Sidnev
Peter the Great St.Petersburg Polytechnic University
St.Petersburg, Russia
alex2922@gmail.com

Abstract—The moment generating function completely characterizes the random variable distribution. At the same time, the natural desire of the researcher is to obtain the density and the distribution function as more informative characteristics. The article analyzes popular transition methods from the moment generating function to the probability density function, in particular, the saddlepoint method, the method based on the Heaviside theorem and Padé approximation, a significant number of the inverse Laplace transform numerical methods. Computational complexity and practical feasibility of the methods are investigated for a number of practical problems of the inverse Laplace transform. Recommendations on the optimal method choice are given and the directions for the further development of the topic are suggested as well.

Index Terms—probability density function, moment generating function, numerical Laplace inversion methods, Padé approximation

I. INTRODUCTION

Moment generating function (MGF) is an important statistical characteristic that uniquely describes the random variable distribution. Since the probability density function (PDF) is related to the MGF through the inverse Laplace transform [1]:

\[ f(t) = L^{-1} \left[ M(-s) \right], \]

the computational complexity can be significantly reduced by using the MGF instead of PDF. For example, MGF of the two random variables sum is calculated as the product of the MGF of these variables [1].

Flowgraph models operate MGF to obtain the stochastic process total time distribution. They can be applied in different areas. For example, in reliability theory for the uptime estimation, in medical statistics for the disease progression and the treatment effectiveness analysis [2], [3], for the power system survivability analysis and it’s weak points identification [4].

Padé approximation, Heaviside’s theorem, Saddlepoint approximation and numerical inverse Laplace transform methods are used to obtain the PDF as the MGF inversion (while the MGF is acquired as the result of the flowgraph specific analysis). The transform method must provide an acceptable approximation error with a relatively low computational complexity and must generate non-negative PDF values.

The paper is organised as follows. Section II introduces the MGF transform via Padé approximation, section III presents the Saddlepoint approximation, section IV provides the review of eight numerical inverse Laplace transform methods. In section V we select methods for the software implementation and comparison on the basis of the preliminary analysis. Finally section VI is devoted to the results discussing.

II. PADÉ APPROXIMATION

The Heaviside theorem is suggested to obtain the PDF applying the inverse Laplace transform of the MGF [5], [6]:

\[ L^{-1} \left[ \frac{U(s)}{R(s)} \right] = \sum_{k=1}^{q} \frac{U(\alpha_k)}{P'(\alpha_k)} e^{\alpha_k t}, \]

where \( \alpha_k \) are the roots of the polynomial \( R(s) \).

MGF Padé approximation can be straightforward provided as series:

\[ \sum_{i=0}^{\infty} c_i * x^i = \sum_{j=0}^{p} a_j * x^j \sum_{k=0}^{q} b_k * x^k \]

Padé approximation is applicable to MGF because:

\[ M(s) = \sum_{n=0}^{\infty} \frac{\mu_n}{n!} * s^n, \]

where \( \mu_n \) is n-th moment.

Ren [5] compares three methods: Padé approximation, maximum entropy method and saddlepoint approximation. It is noted that the Padé approximation is more informative and always provides an analytical approximation for the unknown original PDF.

But this approach has a significant disadvantage: Padé approximation requires the high order derivatives computation (because the n-th moment is the n-th derivative of the MGF). This fact leads to great computational complexity often exceeding the hardware possibilities for the high order derivatives calculation in the symbolic form.

III. SADDLEPOINT APPROXIMATION

Ren [5] also considers the technique of saddlepoint approximation for the MGF-PDF transform:

\[ f(t) = \left( \frac{n}{2 * \pi * K''(s)} \right)^{\frac{1}{2}} * \exp(n * (K(s) - st)), \]
where \( K(s) = \log(M(s)), \quad K'(s) = t, \quad n \) is the number of random variables. For one random variable PDF approximation is equal to [7]:

\[
f(t) = \left(2 \ast \pi \ast K''(s)\right)^{-\frac{1}{2}} \ast \exp(K(s) - st)
\]

The error of this approximation is estimated as \( O(n^{-\frac{1}{2}}) \) [5]. Collins [8] notes this method to have less efficiency than the inverse Laplace transform.

IV. INVERSE LAPLACE TRANSFORM METHODS

There are over a hundred different numerical inverse Laplace transform (ILT) methods. Being applied to the various types of functions they differ in computational complexity and approximation accuracy.

Some ILT methods have Gibbs oscillations – an oscillatory error which occurs when a discontinuous function is approximated with Fourier series. Gibbs oscillations do not disappear when more terms are added to the approximation (the "size" of the excess does not decrease vertically, but only horizontally [9]).

A. Abate–Whitt framework

Different ILT numerical methods can be combined within the same mathematical framework. Abate and Whitt propose the unified formula (1) and notice, that such algorithms as Gaver–Stehfest, Talbot and Fourier series method with Euler summation can fit into this framework [10].

Abate–Whitt framework [11]:

\[
f(t) \approx \sum_{k=1}^{N} \frac{\eta_k}{t} F\left(\frac{\beta_k}{t}\right),
\]

where \( t > 0, \) nodes \( \beta_k \) and weights \( \eta_k \) (internal and external scaling constants) are real or complex numbers that depend on \( N, \) but do not depend on the transform \( F \) or the time argument \( t \) [12].

Abate–Whitt framework has low computational complexity, but it is characterized by a deterioration of the approximation on periodic functions at large values of \( t \) [9]. Since PDF is not a periodic function, this disadvantage can be ignored.

Various ILT methods in this framework differ only by approaches to \( \eta_k \) and \( \beta_k \) calculation.

Several methods belonging to the Abate-Whitt framework are considered below.

1) CME: Method is based on the concentrated matrix-exponential distribution [11].

Matrix-exponential distributions of order \( N \) contain positive random variables with PDF \( f(t) = -\alpha A \exp(t), \) at \( t \geq 0, \) where \( \alpha \) is a real row vector of length \( N, \) \( A \) is an \( N \times N \) real matrix, and \( 1 \) is a column vector of ones of size \( N \) [11].

Let \( A \) be diagonalizable with spectral decomposition \( A = \sum_{k=1}^{N} u_k \lambda_k v_k, \) where \( \lambda_k \) are the eigenvalues, \( u_k \) are the right eigenvectors and \( v_k \) are the left eigenvectors of \( A \) with \( v_k u_k = 1, \) then the PDF of the matrix-exponential distribution can be written as:

\[
f(t) = \sum_{k=1}^{N} c_k e^{\lambda_k t},
\]

where \( c_k = -\alpha A u_k v_k 1.\)

A matrix-exponential distribution is called concentrated if its squared coefficient of variation (SCV) is minimal [13]. The density of the matrix-exponential distribution with the minimal SCV is known analytically only for the order \( N < 3. \) Therefore, the matrix-exponential distributions required for the approximation were calculated using numerical optimization by the expression:

\[
f(t) = c e^{-\lambda} \prod_{j=0}^{N-1} \cos^2(\omega t - \phi_j) = \sum_{k=1}^{N} \eta_k e^{-\beta k t},
\]

where \( c, \lambda, \omega, \phi_j \) are positive real values.

Thus, the values \( \eta_k \) and \( \beta_k \) providing the minimum SCV for the matrix-exponential distribution are the coefficients of the Abate-Whitt framework and are obtained from numerical optimization [14].

It is guaranteed that the CME method is free from Gibbs oscillations and preserves monotonicity [11]. Many of the known methods do not have these properties: for example, the \( f_n(t) \) function for the Euler and Gaver–Stehfest methods can take negative values.

Advantages of the method: it does not cause overshoot, maintains monotonicity, is accurate when using floating point arithmetic, the quality of the approximation improves with increasing order [11].

2) Gaver–Stehfest method: Method is based on the three-parameter exponential PDF properties (Gaver method [15]). Stehfest [16] improves method by taking the weighted average of a Gaver approximations sequence for fixed \( t. \)

Gaver–Stehfest algorithm does not use complex numbers; weights and nodes are real numbers and are calculated as follows [10]:

\[
\beta_k = k * \ln(2)
\]

\[
\eta_k = (-1)^{(k + 1)} \ln(2) * \sum_{j=[\frac{k+1}{2}]}^{\min(k, \frac{N}{2})} \frac{(-1)^{j} 2^j}{j^2 (j - 1)! (k - j)! (2j - k)!}
\]

where \( N \) is the number of terms used in the equation (the method is defined only for even \( N). \) Wang in [10] notices that \( N \) is recommended to be taken in the range from 10 to 14.

The main disadvantage of the method is Gibbs oscillations [11].

Abate and Whitt [12] show that Euler and Talbot methods are more efficient than show that Euler and Talbot methods are more efficient than those of Gaver–Stehfest. It is also noted that the algorithm implementation requires high computational accuracy due to manipulations with large numbers [12], [17]. However, the Gaver–Stehfest algorithm has the advantage of using only real numbers.
3) **Euler method**: is an implementation of the Fourier series method using Euler summation to accelerate convergence [18]. Method is applicable only to odd orders and assumes nodes with positive imaginary parts.

Nodes and weights are calculated as follows [11]:
\[
\beta_k = \frac{(N - 1) \ln(10)}{6} + \pi i (k - 1)
\]
\[
\eta_k = 10 \frac{N-1}{\pi} (-1)^k \xi_k,
\]
where \( \xi_1 = \frac{1}{2} \), \( \xi_k = 1, 2 < k < \frac{n+1}{2} \), \( \xi_n = \frac{1}{2} \).

\( \xi_{n-k} = \xi_{n-k+1} + 2^{-\frac{n-1}{2}} \left( \frac{n-1}{k} \right), 1 < k < \frac{n-1}{2} \)


4) **Talbot method**: Method is based on deforming the contour integral in the Bromwich inversion [19]. It assumes nodes with positive imaginary parts and also applies values of \( \beta_k \) with a negative real part, which is usually outside the region of convergence, but there can be an analytic continuation of \( F(s) \) [9].

Nodes and weights are calculated as follows [11]:
\[
\beta_1 = \frac{2N}{5}
\]
\[
\beta_k = \frac{2(k - 1)\pi}{5} \left( \cot \left( \frac{(k - 1)\pi}{N} \right) + i \right)
\]
\[
\eta_1 = \frac{1}{5} e^{\beta_1}
\]
\[
\eta_k = \frac{2}{5} \left[ 1 + i \frac{(k - 1)\pi}{N} \left( 1 + \left[ \cot \left( \frac{(k - 1)\pi}{N} \right) \right]^2 \right) - \right.
\]
\[
\left. - i \cot \left( \frac{(k - 1)\pi}{N} \right) \right] e^{\beta_k}
\]

Method is not free from Gibbs oscillations [11].

5) **Zakian method**: Method is based on Fourier series method with Padé approximation [11].

General formula is [10]:
\[
f(t) = \sum_{k=1}^{N} \text{Re} \{ K_k F(\frac{\alpha_k}{t}) \},
\]
where \( K_k \) and \( \alpha_k \) are real or complex constants. \( N \) represents the number of terms used in the summation. According to [10], [20] a sufficient value of \( N \) in most cases is \( N = 10 \). Coefficients for \( 1 \leq N \leq 10 \) are calculated in [20].

Zakian proposes two methods for choosing coefficients \( K_k \) and \( \alpha_k \). In the first he compares the Laplace transform \( \delta(t, u) \) (a rational function) with the Laplace transform of \( \delta_n(u - t) \) (an exponential function) and chooses the coefficients so that the rational functions are equal to the classical Padé approximations of the exponential function [21]. Another method includes least squares optimization [22].

6) **Hyperbolic kernel approximation method**: Method is based on the Laplace transform inverse kernel approximation by expressions containing hyperbolic functions \( \sinh \) and \( \cosh \) or their combinations [23].

The general formula [24] is based on the combining two approaches to the Laplace transform inverse kernel approximation \( e^{st} \approx \frac{e^a}{\sinh(a-st)} \) and \( e^{st} \approx \frac{e^a}{\cosh(a-st)} \) to increase accuracy:
\[
f(t) = \frac{e^a}{2t} \left( \frac{1}{2} F \left( \frac{a}{t} \right) + \sum_{n=1}^{N} (-1)^n \left( \text{Re} \left( F \left( \frac{a}{t} + i n \frac{\pi}{t} \right) \right) + \text{Im} \left( F \left( \frac{a}{t} + i (n - \frac{1}{2}) \frac{\pi}{t} \right) \right) \right) \right)
\]

The absolute error depends on the \( a \) as follows [24]:
\[
\varepsilon_{aM} \approx M e^{-4a},
\]
where \( M \) is the maximum absolute value of the original \( f(t) \).

The parameter \( a \) is had to be found empirically. It is recommended to have \( a \) in the following range \( 2 \leq a \leq 6 \) [23], [25]. Brančik and Smith [26] also notice that one of the ways to improve the accuracy of this approximation is to increase the parameter \( a \).

B. **Post–Widder method**

ILT is calculated by the formula [27]:
\[
f(t) := \left( -1 \right)^{N-1} \frac{\left( \frac{N}{N-1} \right)}{N} F^{(N-1)} \left( \frac{N}{t} \right),
\]
where \( F^{(n-1)}(x) \) is the \( n \)-th derivative of \( F(x) \).

The main issue is that method requires high order derivatives computation and is characterized by slow convergence [27].

Post-Widder method does not cause the overshoot and maintains monotonicity [11]. However, according to Horvath’s research [11], the CME method gives better approximation. Davies [28] also notices that the method rarely gives a high accuracy of the approximation.

C. **Laguerre method**

Method is based on Laguerre polynomials. ILT is calculated by the formula [29]:
\[
f(t) \approx \sum_{n=0}^{\infty} q_n l_n(t),
\]
where \( l_n \) is Laguerre function, \( q_n \) are Laguerre coefficients, dependent on \( F \).

Laguerre function:
\[
l_n(x) = e^{-\frac{x}{2}} L_n(x),
\]
where \( L_n \) is Laguerre polynomial.

Laguerre polynomial:
\[
L_n(x) = \sum_{m=0}^{n} \binom{n}{m} \frac{(-x)^m}{m!}
\]

where \( \binom{n}{m} \) is the binomial coefficient.
Laguerre coefficients generating function:

\[ Q(z) = \sum_{n=0}^{\infty} q_n z^n = \frac{1}{1 - z} F\left(\frac{1 + z}{2(1 - z)}\right) \]  

(8)

Formulas 5 – 8 can be rewritten as follows [11]:

\[ f(t) = \sum_{j=0}^{N-1} F^{(j)}(s) \frac{1}{2} \sum_{k=j}^{N-1} \frac{k!}{(j!)^2(k-j)!} l_k(t), \]  

(9)

where \( F^{(j)}(s) \) is the j-th derivative of \( F(s) \).

Davies [28] notices that Laguerre polynomials give an accurate approximation over a wide range of functions.

The disadvantage of this method is that computation requires high order differentiation. Laguerre method also suffers from Gibbs oscillations [11].

D. Other methods

Cohen method is based on the power series \( F(s) \) as a function \( \frac{1}{s} \). However, this method is not applicable when \( F(s) \) has a pole at 0, and the approximation also gets rapidly worse for larger values of \( t \) [11].

Honig-Hirdes and de Hoog methods are the variation of the Fourier series method. Wellekens uses the Fourier series method based on the Padé approximation [11]. Schapery method is described in [30], but according to [28], it rarely gives the high accuracy of the approximation.

V. CHOOSING METHODS FOR THE IMPLEMENTATION

We will determine the most promising solutions for the considered methods.

Padé approximation. Post-Widder and Laguerre methods provide high computational complexity, because of calculating high order derivatives.

Gaver–Stehfest, Euler, Talbot methods suffer from the Gibbs oscillations, and since PDF is restricted to have negative values, these methods are also beyond our consideration.

So we choose four of the most promising methods for the comparison: the saddlepoint approximation, CME method, Zakian method and hyperbolic kernel approximation method.

VI. IMPLEMENTATION AND COMPARISON OF THE SELECTED METHODS

The selected methods are implemented as Matlab functions (Matlab R2020a version was used), because Matlab [31] has one of the most powerful symbolic toolboxes. For the comparison with other methods the CME method is taken in the format suggested in [32] with some parameters pre-calculated by the authors of the method. The saddlepoint method, Zakian method, hyperbolic kernel approximation method are Matlab implemented according to [33]. Zakian method is realised with pre-calculated parameters \( K_i \) and \( \alpha_i \), borrowed from [20]. The \( a \) parameter value is selected empirically for the hyperbolic kernel approximation method. Since Matlab has the built-in inverse Laplace transform function, it is reasonable to compare it with the selected methods.

The method’s average running time is stated as the performance criteria. The hardware in use is Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz, RAM 8.00 GB with Windows 10.

The accuracy of the methods tested is assessed by two parameters: maximum absolute deviation \( \varepsilon_{abs} \), to check for the sharp exceeds in the approximation, and 1-norm of the numerical error \( \varepsilon_n \) calculated by the formula:

\[ \varepsilon_n = \int_0^T |f(t) - f_{appr}(t)| dt \approx \frac{1}{M} \sum_{m=1}^{M} |f(m) - f_{appr}(m)| \]

Original function \( f(t) \) is compared with approximated PDF \( f_{appr}(t) \) at \( M = 200 \) equidistant points.

The problem of the PDF mining from the flowgraph is stated for the beforehand known PDFs just for the testing purpose. The methods are checked on the approximation of exponential, normal, triangular and uniform distributions with well known Laplace transforms. The flowgraph with 5 nodes shown on fig. 1 is also investigated on the subject of the transition time distribution from the node 1 to the node 5. The flowgraph parameters are presented in the Table I. It should be noted that for this graph the 10-th MGF derivative calculation takes 2 seconds, the 12-th derivative calculation takes 4.5 seconds. It confirms that methods based on the high order derivatives calculation are not suitable for the implementation.

![Flowgraph](image)

**Fig. 1. Flowgraph**

**TABLE I**

<table>
<thead>
<tr>
<th>Edge</th>
<th>Probability</th>
<th>Distribution</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>q12</td>
<td>0.2</td>
<td>Exponential</td>
<td>( \lambda = 0.2 )</td>
</tr>
<tr>
<td>q14</td>
<td>0.8</td>
<td>Uniform</td>
<td>( a = 2.4, b = 9.6 )</td>
</tr>
<tr>
<td>q23</td>
<td>1</td>
<td>Triangular</td>
<td>( a = 1.6, b = 6.4, c = 4 )</td>
</tr>
<tr>
<td>q34</td>
<td>0.9</td>
<td>Exponential</td>
<td>( \lambda = 0.5 )</td>
</tr>
<tr>
<td>q35</td>
<td>0.1</td>
<td>Uniform</td>
<td>( a = 6, b = 21 )</td>
</tr>
<tr>
<td>q43</td>
<td>0.5</td>
<td>Triangular</td>
<td>( a = 4, b = 16, c = 10 )</td>
</tr>
<tr>
<td>q45</td>
<td>0.5</td>
<td>Exponential</td>
<td>( \lambda = 0.3 )</td>
</tr>
</tbody>
</table>

A. Exponential distribution

Built-in Matlab function demonstrates the best performance and the highest approximation accuracy (tables II - III) for the exponential distribution (\( \lambda = 0.5 \)) among all the methods in question. A visual comparison (Fig. 2) confirms these results.
TABLE II
EXPONENTIAL DISTRIBUTION APPROXIMATION TIME

<table>
<thead>
<tr>
<th>Order</th>
<th>Matlab function, s</th>
<th>Zakian, s</th>
<th>Saddlepoint, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.0110 ± 0.0005</td>
<td>0.055 ± 0.005</td>
<td>0.111 ± 0.002</td>
</tr>
<tr>
<td>5</td>
<td>0.319 ± 0.009</td>
<td>0.098 ± 0.004</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.356 ± 0.009</td>
<td>0.182 ± 0.012</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.406 ± 0.007</td>
<td>0.353 ± 0.009</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.5015 ± 0.0104</td>
<td>0.860 ± 0.012</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.888 ± 0.019</td>
<td>1.714 ± 0.021</td>
<td></td>
</tr>
</tbody>
</table>

TABLE III
EXPONENTIAL DISTRIBUTION APPROXIMATION ERRORS

<table>
<thead>
<tr>
<th>Order</th>
<th>Matlab function</th>
<th>Zakian</th>
<th>Saddlepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.51E-03</td>
<td>6.81E-03</td>
<td>3.25E-04</td>
</tr>
<tr>
<td>5</td>
<td>3.40E-04</td>
<td>3.70E-03</td>
<td>2.68E-03</td>
</tr>
<tr>
<td>10</td>
<td>5.62E-05</td>
<td>8.02E-04</td>
<td>1.62E-03</td>
</tr>
<tr>
<td>20</td>
<td>9.06E-07</td>
<td>2.30E-05</td>
<td>8.29E-04</td>
</tr>
<tr>
<td>50</td>
<td>3.60E-07</td>
<td>9.00E-06</td>
<td>5.54E-04</td>
</tr>
</tbody>
</table>

B. Normal distribution

The Matlab built-in function is unable to provide the inverse Laplace transform of the normal (Gauss) distribution with the expected value $\mu = 30$ and standard deviation $\sigma = 3$. It returns an unevaluated call to ilaplace function. Saddlepoint approximation provides the set of values that exactly approximates the Gauss distribution only for the bounded segment $0;116$. Zakian and CME methods show sharp spikes in the values of the approximated function. Only the hyperbolic method gives a sufficiently accurate approximation (Table V). A visual comparison shows that CME method is characterized by the sharp exceed of values in small $t$, but Zakian method cannot provide an accurate approximation for $t > 30$ (Fig. 3). Zakian method ensures maximum performance, but because of low accuracy, we recommend to use hyperbolic approximation for this distribution.

TABLE IV
NORMAL DISTRIBUTION APPROXIMATION TIME

<table>
<thead>
<tr>
<th>Order</th>
<th>Matlab function, s</th>
<th>Zakian, s</th>
<th>Saddlepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.072 ± 0.004</td>
<td>0.112 ± 0.004</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.319 ± 0.005</td>
<td>0.096 ± 0.003</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.376 ± 0.014</td>
<td>0.219 ± 0.009</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.44 ± 0.002</td>
<td>0.420 ± 0.018</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>2.92 ± 0.03</td>
<td>2.955 ± 0.009</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>4.55 ± 0.06</td>
<td>5.89 ± 0.02</td>
<td></td>
</tr>
</tbody>
</table>

TABLE V
NORMAL DISTRIBUTION APPROXIMATION ERRORS

<table>
<thead>
<tr>
<th>Order</th>
<th>Matlab function</th>
<th>Zakian</th>
<th>Saddlepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.94E-03</td>
<td>6.03E-02</td>
<td>5.03E-03</td>
</tr>
<tr>
<td>10</td>
<td>1.75E+00</td>
<td>3.49E+02</td>
<td>8.91E-04</td>
</tr>
<tr>
<td>20</td>
<td>p. inf.</td>
<td>p. inf.</td>
<td>4.66E-05</td>
</tr>
<tr>
<td>50</td>
<td>p. inf.</td>
<td>p. inf.</td>
<td>1.85E-05</td>
</tr>
<tr>
<td>100</td>
<td>p. inf.</td>
<td>p. inf.</td>
<td>2.44E-03</td>
</tr>
</tbody>
</table>

*p. inf. (practically infinite) stands for values larger than 1000

C. Triangular distribution

The saddlepoint approximation for the triangular distribution with parameters $a = 10; b = 50; c = 30$ is unable to find the answer: method fails with error "Unable to find..."
The highest approximation accuracy and performance is demonstrated by the built-in Matlab function (Tables VI, VII). Zakian method does not provide the required approximation accuracy (Fig. 4).

### TABLE VI
**Triangular distribution approximation time**

<table>
<thead>
<tr>
<th>Matlab function, s</th>
<th>Zakian, s</th>
<th>Saddlepoint, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0108 ± 0.0005</td>
<td>0.16 ± 0.03</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>CME, s</th>
<th>Hyperbolic, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.46 ± 0.08</td>
<td>0.144 ± 0.005</td>
</tr>
<tr>
<td>10</td>
<td>0.68 ± 0.02</td>
<td>2.56 ± 0.03</td>
</tr>
<tr>
<td>20</td>
<td>1.068 ± 0.018</td>
<td>5.03 ± 0.04</td>
</tr>
<tr>
<td>50</td>
<td>2.34 ± 0.08</td>
<td>13 ± 1</td>
</tr>
<tr>
<td>100</td>
<td>3.39 ± 0.04</td>
<td>25.56 ± 0.17</td>
</tr>
</tbody>
</table>

### TABLE VII
**Triangular distribution approximation errors**

<table>
<thead>
<tr>
<th>Matlab function</th>
<th>Zakian</th>
<th>Saddlepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon_n$</td>
<td>$\varepsilon_{abs}$</td>
<td>$\varepsilon_n$</td>
</tr>
<tr>
<td>2.50E-04</td>
<td>5.00E-02</td>
<td>1.17E-03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>CME</th>
<th>Hyperbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9.24E-04</td>
<td>5.92E-02</td>
</tr>
<tr>
<td>10</td>
<td>4.08E-04</td>
<td>5.43E-02</td>
</tr>
<tr>
<td>20</td>
<td>2.88E-04</td>
<td>5.20E-02</td>
</tr>
<tr>
<td>50</td>
<td>2.58E-04</td>
<td>5.05E-02</td>
</tr>
<tr>
<td>100</td>
<td>2.55E-04</td>
<td>5.05E-02</td>
</tr>
</tbody>
</table>

### TABLE VIII
**Uniform distribution approximation time**

<table>
<thead>
<tr>
<th>Matlab function, s</th>
<th>Zakian, s</th>
<th>Saddlepoint, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0114 ± 0.0016</td>
<td>0.18 ± 0.08</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>CME, s</th>
<th>Hyperbolic, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.331 ± 0.004</td>
<td>0.21 ± 0.08</td>
</tr>
<tr>
<td>10</td>
<td>0.6 ± 0.10</td>
<td>0.42 ± 0.16</td>
</tr>
<tr>
<td>20</td>
<td>0.83 ± 0.03</td>
<td>3.55 ± 0.03</td>
</tr>
<tr>
<td>50</td>
<td>1.70 ± 0.07</td>
<td>10.2 ± 0.2</td>
</tr>
<tr>
<td>100</td>
<td>2.98 ± 0.11</td>
<td>21.6 ± 0.8</td>
</tr>
</tbody>
</table>

### TABLE IX
**Uniform distribution approximation errors**

<table>
<thead>
<tr>
<th>Matlab function</th>
<th>Zakian</th>
<th>Saddlepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon_n$</td>
<td>$\varepsilon_{abs}$</td>
<td>$\varepsilon_n$</td>
</tr>
<tr>
<td>2.50E-04</td>
<td>2.50E-02</td>
<td>3.14E-03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>CME</th>
<th>Hyperbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.93E-03</td>
<td>2.56E-02</td>
</tr>
<tr>
<td>10</td>
<td>8.93E-04</td>
<td>2.55E-02</td>
</tr>
<tr>
<td>20</td>
<td>4.45E-04</td>
<td>2.52E-02</td>
</tr>
<tr>
<td>50</td>
<td>2.61E-04</td>
<td>2.51E-02</td>
</tr>
<tr>
<td>100</td>
<td>2.52E-04</td>
<td>2.51E-02</td>
</tr>
</tbody>
</table>

Fig. 4. PDF approximation. Triangular distribution

### D. Uniform distribution

The saddlepoint approximation for the uniform distribution with parameters $a = 20; b = 40$ fails with the error "Unable to find explicit solution". The error for the built-in Matlab function and CME method is explained by the PDF approximation corners rounding (Fig. 5). The CME method provides the best accuracy for the order of $N = 50$ (Table IX). The built-in Matlab function provides the highest performance with the approximation error of the order $10^{-2}$, that makes it the best option among the considered methods. The hyperbolic approximation method suffers from Gibbs oscillations. The Zakian method does not provide the required approximation accuracy.

### E. Flowgraph

For the flowgraph with different transition time distributions (Fig. 1, Table I) the saddlepoint approximation method fails with the error "Unable to find explicit solution" and the same

![PDF](image.png)

Fig. 4. PDF approximation. Triangular distribution

![PDF](image.png)

Fig. 5. PDF approximation. Uniform distribution
result is demonstrated by the built-in Matlab function that returns an unevaluated call to ilaplace function. The Zakian method does not provide the required approximation accuracy (Fig. 6). The best approximation is provided by the CME method (Table XI). For this case the CME method is faster than the hyperbolic approximation and the Zakian method as well (Table X).

### Table X

<table>
<thead>
<tr>
<th>Matlab function, s</th>
<th>Zakian, s</th>
<th>Saddlepoint, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>5.09 ± 0.14</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>CME, s</th>
<th>Hyperbolic, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2.48 ± 0.05</td>
<td>10.20 ± 0.12</td>
</tr>
<tr>
<td>10</td>
<td>4.83 ± 0.11</td>
<td>20.25 ± 0.19</td>
</tr>
<tr>
<td>20</td>
<td>8.77 ± 0.12</td>
<td>32.40 ± 0.18</td>
</tr>
<tr>
<td>50</td>
<td>21.8 ± 0.6</td>
<td>81.50 ± 0.82</td>
</tr>
<tr>
<td>100</td>
<td>34.5 ± 1.6</td>
<td>182 ± 8</td>
</tr>
</tbody>
</table>

### Table XI

<table>
<thead>
<tr>
<th>Matlab function</th>
<th>Zakian</th>
<th>Saddlepoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon_n$</td>
<td>$\varepsilon_{sabs}$</td>
<td>$\varepsilon_n$</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>2.26E-03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Order</th>
<th>CME</th>
<th>Hyperbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5.52E-04</td>
<td>7.08E-03</td>
</tr>
<tr>
<td>10</td>
<td>5.34E-04</td>
<td>7.95E-03</td>
</tr>
<tr>
<td>20</td>
<td>5.67E-04</td>
<td>9.83E-03</td>
</tr>
<tr>
<td>50</td>
<td>5.80E-04</td>
<td>1.05E-02</td>
</tr>
<tr>
<td>100</td>
<td>5.81E-04</td>
<td>1.04E-02</td>
</tr>
</tbody>
</table>

![PDF Approximation](image)

Fig. 6. PDF approximation of the transition time between the first and the last nodes of the graph

### VII. Conclusion

In this paper we consider some well-known MGF inversion methods. Next, we have identified a group of methods that, according to the sources, are characterized by the least computational complexity and the absence of Gibbs oscillations. This group, which includes saddlepoint approximation, CME method, Zakian method and hyperbolic kernel approximation method, was tested on inverse Laplace transform problems for a number of distributions (exponential, normal, triangular, uniform). We also tested these methods on the meaningful problem represented by the five-state flowgraph. All methods are compared with the built-in Matlab function according to the criteria of accuracy and performance. The following conclusions were made on the test results:

- the Zakian method with the recommended in [10], [20] order $N = 10$ failed in most cases. So the calculation of parameters for higher orders is required;
- the hyperbolic kernel approximation method is not free from Gibbs oscillations and also requires the empirical selection of the $\alpha$ parameter, which makes it difficult to use method in the real cases when true distribution is unknown;
- the saddlepoint method gives an exact approximation for exponential and normal distributions, but it unable to provide the acceptable approximation of the PDF for uniform and triangular distributions;
- the built-in Matlab function provides the most accurate and fast result for exponential, triangular and uniform distributions, but does not cope with the MGF transform for the normal distribution and the flowgraph (fig. 1);
- thus, we can recommend the built-in Matlab function or the saddlepoint approximation only for a number of distributions;
- the CME method is the most stable among the considered methods. It is not subject to Gibbs oscillations, uses pre-calculated parameter values, so it has a relatively low computational complexity. In fact the definition of PDF here is reduced to summation and depends only on the complexity of the MGF expression. It is advisable to test the method on flowgraphs with large number of probabilistic transitions.

### References


Generator of automated tools for program instrumentation

Mikhail Onischuck
Peter the Great St.Petersburg Polytechnic University
Saint-Petersburg, Russian Federation
orcid.org/0000-0001-5359-0161

Vladimir Itsykson
Peter the Great St.Petersburg Polytechnic University
Saint-Petersburg, Russian Federation
orcid.org/0000-0003-0276-4517

Abstract. Instrumentation is one of the methods used in dynamic program analysis for assessing software performance. This paper proposes a technology for constructing software instrumentation tools for different programming languages. The instrumentation functions are described within this approach using several grammar-based DSLs. The obtained instrumentation toolkit is the result of generating a new system based on formal descriptions of the instrumentation process. The tracing functions are embedded into the original program using a TXL utility; a conversion program is also generated for this utility. The developed prototype was tested on 4 large projects written in different programming languages: Java, Python, C++ and Object Pascal. The tests confirmed the efficiency of the approach and the applicability of the developed prototype.

Keywords — instrumentation, instrumentation tools generation, formal language grammar, TXL

I. Introduction

Controlling software quality is a major challenge for the software industry. IT companies spend tremendous efforts and resources on diverse methods and practices for software quality assurance. The starting stage generally consists in software quality analysis, checking whether the software is compliant with the specifications and searching for errors. Instrumentation is a mechanism used for software analysis. Instrumentation is commonly understood as inserting additional code into the source program, which allows detecting and monitoring the parameters characterizing the software performance, with options for debugging and troubleshooting [1].

There are different approaches to program instrumentation. Manual instrumentation implies that the developers independently log snippets, relying on their understanding of the program logic. Depending on the goals set in automated instrumentation, the developers use some tool for inserting the logging code into the source program based on certain rules. Unfortunately, most of the existing tools come with limitations restricting the options for code instrumentation. What is more, such tools are generally hard-coded to only work with a specific programming language, even though instrumentation may be required for programs in any language.

In this paper, we propose a universal approach to solving the instrumentation problem incorporating a declarative framework using the grammar of the target programming language, serving to generate automated systems for software instrumentation.

The paper is organized as follows. Section 2 describes the technology we have developed for generating instrumentation systems. Section 3 considers a prototype instrumentation tool generator. Section 4 presents the results of testing the approach on real software projects. Section 5 analyzes the existing solutions in automated instrumentation. The paper is concluded by summarizing the results and outlining the directions for further development.

II. Technology for generating instrumentation tools

In general, depending on the programming language and the runtime environment, either the source code or some intermediate representation (e.g., bytecode) can be instrumented. For example, [2] considers source code instrumentation in C++ for verifying potential vulnerabilities that can threaten the system security. A technology developed in [3] introduces an instrumentation framework based on analysis and processing of an intermediate representation containing some form of an abstract syntax tree. In contrast, [4] and [5] discuss bytecode instrumentation for a Java virtual machine.

Because our study deals with a universal instrumentation system independent of the programming language and runtime environment, the only way to organize this system would have to be source-code instrumentation.

Instrumentation consists in converting the source program into a new one, with special code added to it which enables tracing once the program is executed. This means that the task of instrumentation is reduced to converting (transforming) one source code into another.

A. Methodology

Since one of the requirements to the approach developed is the option to instrument programs written in different languages, it should incorporate a mechanism for generating instrumentation systems for different target languages. An obvious solution is to use the grammar of the target language as input for the instrumentation system. Since standard grammars are not originally designed for instrumentation, a grammar markup mechanism should be developed, adding semantics to the grammar for subsequent use in instrumentation rules. We achieve this by utilizing grammar annotations, created once for each target programming language. The actual instrumentation rules are developed by the user solving a specific applied problem.

The general schematic for the approach developed is shown in Fig. 1.
The module for generating the transformation rules takes as input the grammar of the target language, the grammar annotation for instrumentation, and the instrumentation rules; the module then generates transformation rules based on these data, feeding them to the input of the program transformation module together with the original source code. The program transformation module in turn generates an instrumented program semantically equivalent to the original one with the added instrumentation code.

B. Representation and processing of program source code

Source code transformation has been examined in great detail; a wide range of tools have been developed that can effectively solve this task using some form of a parse tree or a concrete syntax tree (CST) for internal representation. The nodes of this tree represent syntactic structures and individual elements adopted in the target programming language, so making changes to the processed code (its transformation) can be distinguished as a separate stage of instrumentation (see Fig. 1). Such systems require a description of transformations at the level of individual parse tree nodes (and/or sets of nodes) to operate. While it can be difficult to understand this description or correlate it with the problem solved by the user, an additional complication is that it should strictly follow the grammar of the language used. These drawbacks lead us to identify another significant stage of instrumentation that is preparing for changes (generating a description of the transformations). The description of the changes can be simplified by introducing some primary representation with limited functions but at the same time universal enough to be used for processing source code in different programming languages. Because there are no generally accepted standards for grammar formatting and structuring, the above requirement means that the instructions describing the purpose of changes (instrumentation rules) should be separated from the instructions describing repeating primitive operations specific to the given grammar and the syntax elements used (grammar annotation, Fig. 1).

Taking CST as their internal representation, the systems for source code transformation process syntactic structures by moving from higher (file, module, expression set) to lower levels (string literals, digits). The order in which CST nodes are visited and whether the same node can be revisited depends solely on the specifics of a particular tool and the required transformations. Therefore, the instrumentation task can be solved using a method for step-by-step recursive descent down the parse tree to be modified by inserting additional code (the top-down one-pass method).

Let us define a workspace where instrumentation will be done, i.e., the working context that is some subset of CST nodes given as $Y = \{g \in G \mid A(g)\}$, where $A(g)$ is the condition (logical statement about the properties) that the element $g$ belongs to some subset $Y$, while $G$ is the entire set of nodes of a particular parse tree. At the same time, using a CST means using a large number of intermediate nodes in accordance with the grammar rules used constructing it. Consequently, information should be collected during the top-down traversal of the parse tree (a), to be subsequently used for (b) making a decision whether a node belongs to the instrumentation context before actually inserting the code.

Thus, we have formulated a method for processing parse tree nodes and a mechanism for controlling the workspace of this process.

C. Transformation of program source code

The source code of programs written in a wide range of programming languages can be processed either by developing specific tool finely tailored to the individual languages, or taking an existing tool or language (with the appropriate runtime environment), such as, for example, “The Meta-Environment” [6], Stratego/XT [7], Rascal [8], TXL [9] or similar. We have chosen the TXL language to confirm the applicability of the approach.

TXL is a domain-specific language designed to support source analysis and processing through rule-based structural transformation [9]. The FreeTXL compiler/interpreter (referred to as the TXL utility from now on) is the official runtime environment for programs in the TXL language.

The TXL language is relatively simple and pure; furthermore, the TXL utility offers such benefits as binary executables available for various software platforms and easy integration (providing XML output). For these reasons, we decided to use the TXL utility for transformation over the source code which is in turn produced by the instrumentation system generator. In view of this, the generated 'instrumentation system' is understood in this study as the description of transformations in the TXL language together with the grammar of the target programming language and the TXL runtime environment.

The where clause is used in TXL to constrain the composition for transformation rules and functions [10], while sequences of such expressions are combined by means of conjunction. On the other hand, applying logical disjunction to comparison operators from the start within this syntactic structure allows introducing expressions in conjunctive normal form (CNF), obtained from predicates constructed by users to describe instrumentation contexts.

Due to the functional nature of the TXL language, we decided to use the arguments/parameters of the chain of functions to pass the nodes collected during the traversal down the parse tree. In this case, the “chain” is a particular solution to the instrumentation problem, accounting for the above one-pass method for CST traversal.
D. Grammar annotation

It can be a challenge to extract the directed acyclic graph (DAG) representing the nested syntactic hierarchy from the grammar of an arbitrary programming language. This obstacle can be overcome by either manual grammar marking up or by constructing heuristics sufficient for the purpose of defining the scope of the instrumentation on the parse tree. For this purpose, we constructed an XML-based format describing grammar annotations, including such information as:

- description of the syntactic structures of the target programming language that are the most significant for the end user in accordance with the grammar, also including:
  - a text template describing the type of structure;
  - instrumentation points combined with a simplified instrumentation algorithm;
- directed acyclic graph of the hierarchy of nesting syntactic structures;
- points of interest describing the text data to be retrieved from the parse tree nodes;
- auxiliary user-defined functions in the TXL language.

E. User description of the instrumentation process

The prototype uses a declarative domain-specific language (DSL) to describe user-defined rules; it is based on the languages of such projects as Annotation File Utilities [11] and AspectJ [12]. Fig. 2 shows an example of the rules described using the developed language from the system's end user perspective, aimed at logging the first if-statement executing in the context of the "main" method contained in the "Main" class.

![Fig. 2. End user description of the instrumentation rules.](image)

The following main components comprise the user description of the set of rules are (the number of the list item corresponds to the circled number in the figure):

1) listing the source code fragments used in this set of rules and their relative file paths;
2) listing the instrumentation contexts of interest to the user (both simple, i.e., a set of statements about the properties of a syntactic structure, and composite, i.e., several contexts joined by first-order logical operators);
3) grouping the instrumentation steps as named rules;
4) refining the instrumentation context using programming language keywords, modifiers (enclosed in square brackets) and text patterns, if any are required for the task to be solved by the user;
5) setting specific instrumentation points by their identifiers;
6) creating user-defined variables from text elements and constant values;
7) a namelist of fragments to be inserted simultaneously into the same place, specifying the parameters, if any are required according to the text of the snippet used.

The main benefits of the DSL developed is that it provides two methods for describing instrumentation contexts (item 2) and an option for successively refining the context specified (items 4, 5). This way, the user can considerably limit the workspace for transformations while the implementation details remain concealed. Nevertheless, the transformation descriptions output by the generator can be used as the initial step for more complex instrumentation routines.

F. Tool generation procedure

The input artifacts for the “generator and transformation tool” system are the following:

- source code of the program to be instrumented;
- grammar description of the target language that the source code is written in;
- grammar annotation;
- description of user-defined instrumentation rules;
- source code snippets (i.e., “fragments”) in the target programming language to be inserted;
- additional startup and runtime environment parameters.

As an intermediate output, the generator provides the instructions for the transformations to be performed with a specific input file with the source code in accordance with the grammar given for the target programming language.

The output artifact is the source code in the target programming language, which has been subjected to the required transformations.

The developed prototype automatically generates the transformation instructions as a set of interconnected TXL functions in the following order:

1) load, parse and check the dependences of the source code fragments used in accordance with the rules described by the end user;
2) calculate the maximum distances from the root node for each node of the DAG representing the key structures of the target language in accordance with the grammar annotation provided;
3) build wrappers over standard comparison operators;
4) build functions for implementing the tasks assigned to the points of interest;
5) build functions checking whether CST nodes belong to contexts;
6) build function chains in accordance with instrumentation rules allowing for the user's requirements;
7) build auxiliary TXL functions;
8) build user-defined functions;
9) build the main TXL function and apply policies for additional user-defined functions;
10) update the states of functions that are chain elements;
11) generate TXL instructions for the required transformations and call the TXL utility.

The general structure of a typical chain of calls to programmer-defined domain-specific functions:

1) C-functions (collect) are rule-type TXL functions, designed to accumulate information from the parse tree nodes. This information is later used to assess whether the node belongs to the chosen instrumentation context. Such functions operate by calling the next function from the chain and passing it all the values of the arguments that were received by the current function, together with the node considered.
2) F-function (filter) is a function designed to filter CST nodes relative to contexts described by the end user by calling the auxiliary function for assessing whether the node belongs to the context and passing it the collected nodes.
3) R-functions (refine) are functions designed to refine the context to a limited subset of some required syntactic structures of the target programming language in accordance with one of the implemented modifiers:
   a) "first" searches and processes only the first encountered node from the current subtree in accordance with the type specified in the grammar annotation;
   b) "all" searches and processes all nodes in accordance with the type specified in the annotation;
   c) "level" searches and processes the nodes located at the same (first) nesting level. A schematic example illustrating how this modifier works is given in Fig. 3: different nesting levels of syntactic structures are shown from the bottom-up, a sequence of structures from the standpoint of source code is shown from left to right; the colors correspond to different types of CST nodes (nodes of the required type are colored in red; nodes to be processed are colored in dark red); the numbers indicate the order in which the pass is performed in the TXL environment.
4) I-function (instrument) are functions designed directly for instrumentation in accordance with the patterns (search and replace) specified in the annotation and the operation algorithm.

Function chains should be generated for each individual expression containing a context refinement and the keyword "add" (see Fig. 2) together with a list of code fragments, as a separate rule (group of refinements) in accordance with the description order.

III. Prototype implementation

To test the efficiency of the above approach, we constructed a prototype instrumentation tools generator combining the generator application built based on the TinyXML2 [13] and Boost [14] libraries in C++ for the purpose of parsing grammar annotations along with user descriptions and interaction with the TXL utility, respectively, and the application itself. Fig. 4 shows the general schematic for the prototype together with the artifacts necessary to solve the problem posed: in accordance with the initial model (Fig. 1), the generator utility is a module producing an intermediate (optionally cached) description of instrumentation instructions in a format that can be used by the second part of the two, i.e., the transformation system. The generator utility acts as a module producing the transformation rules in this case, while the TXL utility performs the function of the program transformation module. The colors of the arrows in the figure correspond to different frequencies of analysis and processing of artifacts by the generator and the TXL utility (orange is more frequent, purple is less frequent), the colors of the input artifacts characterize the degree to which it is difficult for the user to create them (red is very difficult, blue is moderately difficult, green is easy).
The applicability of the developed instrumentation method and the functionality of the implemented prototype were verified using several industrial open source projects. The projects and the source code samples were chosen based on the capabilities of the TXL grammars available at the time of this study [15]; notably, some elements of the grammars were slightly modified to better suit the described instrumentation approach (increased separation of syntactic structures). As a result, four projects were chosen for the experiments, written in four different programming languages:

- AspectJ [12] is a system and DSL designed to implement aspect-oriented programming principles within the Java language. Version 1.9.5 was chosen for the experiment.
- Keras library [16] is an add-on library for high-level processing and construction of deep learning neural network models for the Python language. Version 2.3.1 was chosen for the experiment.
- Boost library [14] is a multifunctional modular library for building software products using the C++ language. Version 1.72.0 was chosen for the experiment.
- Lazarus IDE [17] is a graphical cross-platform environment for rapid application development in the Object Pascal language and its dialects. Version 2.0.8 was chosen for the experiment.

Different syntactic structures of the languages were instrumented in the experiments, such as import statements, class bodies, methods, branch operators, and loops, in particular using such means as different nesting levels of programming language structures. The source codes of the performed experiments are available in [18].

The approach was found to be efficient the most for instrumenting an AspectJ project written in Java. As for other projects, we found both minor limitations associated with multivariate forms of some syntactic structures (for example, the import statement in Python), and major difficulties when the descriptive capabilities of the grammars of languages provided by the TXL developers and/or the community were found to be insufficient. In particular, the Object Pascal grammar covered about 80% of the code base of the Lazarus IDE project at the time of the study, while the C++ grammar (specifically designed for the older version of the C++ standard) covered only 21% of the Boost library base. Simplified language grammars should be further refined for the developed prototype to be used industrially, providing full support for the standards of these languages.

We can conclude from our findings that the proposed approach and the developed prototype generator are largely applicable for the tasks described. There are certain limitations because existing grammars of the programming languages are imperfect; moreover, the prototype constructed has some drawbacks yet to be eliminated.

V. Comparison with counterparts

There is a wide range of different software systems offering options for automating the instrumentation process to some degree. Examples of such systems include tools for test coverage analysis (GCC Gcov [19], Froglogic Squish Coco Coverage Scanner [20]), code analysis (Testwell CTC++ Preprocessor [21], Bullseye Coverage [22]), tracing and statistics calculations (Google Web Tracing Framework [23]), vulnerability assessment (see [2] and [4]).

Each of these projects only works with a very small subset of programming languages, and they need to be further developed and adapted to work with new languages. For open source projects, this can be done by forking the main code base but this takes a lot of time and resources. Commercial tools (Testwell CTC++, Bullseye Coverage and Froglogic Squish Coco) can only be expanded by the developers.

Compared to such instrumentation methods as, for example, bytecode processing [5], application of the aspect-oriented programming paradigm [24], or reduction to a single intermediate representation with subsequent reconstruction [3], the main difference of the approach proposed our study is that the user can flexibly control the instrumentation process and adapt the instrumentation for other programming languages by specifying instrumentation rules in terms of the target programming language, also independently creating and annotating grammars in terms of parsing texts in formal languages.

VI. Conclusion

The study presents an approach to software instrumentation, with a prototype developed for a generator for automated instrumentation tools, for which the DSL of instrumentation rules and the format for writing annotations for formal grammars were described. We tested the prototype on several industrial open-source projects, confirming that it was functioning properly and that the approach could be applied successfully.

If a program is represented as a text in some formal language with a developed grammar describing the structures of this language, this approach can be considered sufficiently universal for solving the instrumentation problem. This, however, implies that the capabilities of such a generator mainly depend on the capabilities of the transformation system applied and the grammar of the target language, which was confirmed experimentally. High performance is the most crucial factor for program execution: it can be achieved by making the program as close as possible to a machine-generated semi-structured format, removing most of the information that does not improve this indicator (i.e., information about the original structure of the program). All of this limits the potential applications of the approach to an environment with access to structural information about the program.

The technique can be further developed by expanding the DSLs constructed, which can allow overcoming the existing context constraints, and conducting more focused experimental studies for a wider range of programming languages.
VII. References


Visualizing Russian kinship term possessive sequences as family trees

Anna Golub
Faculty of Infocommunication Technologies
ITMO University
St. Petersburg, Russia
anna.golub.923@gmail.com

Gleb Bondarenko
Faculty of Infocommunication Technologies
ITMO University
St. Petersburg, Russia
bondoll2001@mail.ru

Alyona Belova
Faculty of Infocommunication Technologies
ITMO University
St. Petersburg, Russia
belova.alyona.itmo@gmail.com

Darya Karmaz
Faculty of Infocommunication Technologies
ITMO University
St. Petersburg, Russia
dasha.karmaz@yandex.ru

Abstract—As frequently as they are encountered in texts of various genres, Russian kinship term possessive sequences remain confusing even for the native speakers. The paper presents the authors’ original computer science project, whose goal was to suggest a method of extracting such word sequences from a piece of text and visualizing them in an easily comprehensible way. Such an attempt of kinship relations analysis automatization might contribute to future research in history, linguistics, and literary studies and be of use to those studying Russian as a foreign language.

Keywords—kinship term, possessive structure, family tree, natural language processing, Russian

I. INTRODUCTION

Russian possessive sequences including several kinship terms (e. g. сестра мужа моей матери — my mother-in-law’s sister, бабушка шурин его брата — his brother’s brother-in-law’s grandmother) often appear confusing in written text as well as in oral speech since it is difficult to quickly calculate the relations between the relatives mentioned. Besides, such phrases often include names of relatives by marriage (мать матери — a wife’s mother, шурин — a wife’s brother, etc.). Those kinship terms are becoming increasingly obsolescent in modern Russian as they only constitute for 1.4 per cent of all kinship term entries in Russian National Corpus [1] in 1990—2020. Therefore, appearing in a sequence, they make it even harder to comprehend.

The goal of the authors’ original computer science project was to create a computational tool that would extract the sequences in question from a given piece of text and visualize them in an easily comprehensible way. Such an attempt of kinship relations analysis automatization might contribute to future research in history, linguistics, and literary studies, e. g. scientific analysis and systematization of fiction and memoirs. Moreover, this technology might be utilized by those studying Russian as a foreign language in order to ease the process of Russian kinship terms’ meaning comprehension and memorization.

The following text processing stages were suggested for the computational tool:

1. finding kinship term possessive sequences in the given text;
2. analyzing the family relations that the each of the sequences presents and building graphs upon them;
3. visualizing the graphs as family trees using the existing tools for data visualization.

II. STATE OF THE ART

A. Kinship Term Possessive Sequences

As to our knowledge, a comprehensive solution to the problem has not yet been suggested. A great deal of research has been done on kinship term systems and the grammar of possession in various languages. However, possessive structures with kinship terms have not been subject to in-depth investigation. A few researchers touched on the topic of kinship terms in their studies of possessive structures (Dahl & Koptjevskaja-Tamm 2001 [2], Paykin & Van Peteghem 2003 [3], Jones 2010 [4]). Unfortunately, the approach taken was exclusively theoretical, and in addition, the only type of phrases discussed was those consisting of a possessive pronoun and a single kinship term, such as her sister. Aside from that, some papers describing data collected through fieldwork mention kinship term possessive sequences, but that kind of research does not seem applicable to building a tool for their computational processing.

B. Family Tree Visualization

In terms of visualization, there are many software tools that are suitable for depicting family trees. To begin with, specialized packages (e. g. ggenealogy [5]) provide plotting methods for genealogical data. However, the hierarchical nature of a structure does not allow horizontal edges and node skipping, which are required for valid representation of relationships in kinship term possessive sequences. This is accurate for the libraries with more general visualizing functionality (e. g. Graphviz [6], Plotly [7], Toytree [8]) as well. Furthermore, these visualization tools have rather limited customization options while the goal was to display confusing lineages in the most efficient way.

Additionally, to our knowledge, some other family tree visualization tools appropriate for the assigned task [9] are unfortunately unavailable for public use.
Moreover, there is genealogy software with intuitive family tree builders for illustrating pedigrees (e.g. Family Historian, Legacy Family Tree, etc. [10]). Despite the representation of kinship relationships in a comprehensive, visually organized manner, these applications require user interaction, which makes automatic visualization significantly more complicated.

Taking all of the aforementioned into account, we decided to use NetworkX [11] and Matplotlib [12] to develop our own visualization algorithm best fitting the requirements.

III. SOLUTION

A. Text Search

All the source code is written in Python. At first, using NLTK, a Python library for natural language processing [13], the given piece of text is split into sentences; then each of them is tokenized into words. Next, employing pymorphy2 [14] for part-of-speech tagging and further morphological analysis, continuous word sequences are extracted from sentences. At this point the sequences consist of:

- one or more **kinship terms**, the first one of them in any case while all the rest in the genitive case exclusively;
- certain kinds of **modifiers**, namely long- or short-formed adjectives and participles, ordinal numerals and adjective pronouns;
- not more than one non-kinship **noun** in the genitive case. If included, this word is the last one in the sequence (see sequence type 4 below).

Afterwards, each of the sequences is recognized as one of the sequence types listed below, which depict most instances of kinship term possessive sequences in Russian. The GEN abbreviation stands for the genitive case while \( n \) signifies the possible number of the word’s occurrences.

1. **possessive adjective / possessive pronoun** + **kinship term (GEN)**
   
   Example: бабушкиному мужу
   
   ‘grandma’s husband’

2. **kinship term** + **kinship term (GEN) \( n = 0,1,2.. \)** + **possessive adjective / possessive pronoun** + **kinship term (GEN)**
   
   Example: муж бабушки моей сестры
   
   ‘my sister’s grandma’s husband’

3. **kinship term** + **kinship term (GEN) \( n = 0,1,2.. \)** + **possessive adjective / possessive pronoun** + **kinship term (GEN)**
   
   Example: муж бабушки моей сестры
   
   ‘my sister’s grandma’s husband’

4. **kinship term** + **kinship term (GEN) \( n = 0,1,2.. \)** + **noun (non-kinship) (GEN)**
   
   Example: мужем бабушки подруги
   
   ‘friend’s grandma’s husband’

5. **kinship term** + **kinship term (GEN) \( n = 0,1,2.. \)**
   
   Example: моя сестра сестры
   
   ‘sister’s grandma’s husband’

Then, the sequence is cast to the so-called normal form:

- kinship terms are put in the nominative singular form;
- non-kinship nouns are put in the nominative case with the number unchanged;
- possessive adjectives are replaced with their stem nouns in the nominative singular form;
- possessive pronouns are replaced with the corresponding personal ones in the nominative case (the forms of the свой pronoun are substituted by некто).

Afterwards, the sequence is reshuffled so as to put the words in the direct relation order (see examples below). If the sequence then starts with a kinship term, the first-person singular pronoun я is inserted into the beginning.
Examples:

a. бабушкиному мужу — я бабушка муж
   grandma’s husband — me grandma husband

b. мужу бабушки сестры моей — я сестра
   husband grandma sister my — me sister
   бабушка муж
   grandma husband

Thus, all the words in the sequence, except for the first one, turn out to be kinship terms in the nominative singular form. Preprocessed this way, the sequence is fit for further analysis.

B. Kinship Relations Analysis

For each word in the sequence, except for the first one, the following actions are performed.

First, a graph fragment template is uploaded. The template presents the relationship between this and the previous character in the sequence. Here, by character we mean any relative in the chain of connections described by the sequence. Within the template all the characters are connected directly, either through parent—child or wife—husband type of connection. See the сестра (sister) template below as an example.

Next, the template is incorporated in the graph that has been built so far, namely the root of this template is aligned with the top of the previous one. Here, by the template top we mean the character signified by this template’s corresponding word. In turn, the template’s root is the character whose relation to the top of the template is being described by the template word. (In the sister template above я (me) is the root while сестра (sister) is the top.) Consequently, by the tree root we mean the root of the first template and by the tree top the top of the last template included in the graph. See example below: the бабушка (grandmother) template being aligned with the сестра (sister) template. (For now, we only discuss the maternal grandmother.)

Finally, if multiple nodes of the graph correspond to the same character of the sequence, they are merged into one. Such cases are unraveled based on the heuristic that each character may only have one mother and one father. The graph above is thus transformed into the following one.

Thus, in the resulting graph all the characters are connected directly to each other. It is worth mentioning that due to linguistic polysemy, the relationship reflected in a kinship term might be described by a few different templates (e. g. in the above example the grandmother might be both maternal and paternal). Consequently, several graph structures are built, considering all the possible template combinations for the kinship terms in the sequence.
C. Graph Visualization

The family trees are drawn using NetworkX and Matplotlib in Python. All the edges of the graph are added to the list in the loop that goes through the characters and their connections. The root node gets zero coordinates, for other nodes the following rules apply:

- parent is positioned one point higher than their child;
- child is positioned one point lower than their parent;
- wife/husband is drawn at the same level and one point to the right from their spouse;
- if the determined spot is already taken, the node is shifted one point to the right.

As to colors, the following rules apply:

- for the tree top and the tree root nodes, blue color is used;
- otherwise, if the character is directly mentioned in the kinship term sequence, the node has a light blue color;
- if there is no direct mention of the character, the node is painted light gray.

The gender is displayed through the shape of the nodes: circles for females, squares for males; a rhombus is used for the root node. As a result, a PNG file presents the graph with a gray background, the kinship term sequence and the original sentence. The file is the final program output. See the visualization of the sequence муж бабушки моей сестры (my sister's grandmother's husband) as an example.

IV. Evaluation

A. Evaluation Process

In order to evaluate the tool’s performance, it was run on a purposefully collected corpus of texts, selected manually from the Russian National Corpus. The corpus consists of 3067 words and includes at least five entries of each of the kinship terms while keeping a rough balance between the sequence types.

For text search evaluation, each kinship term possessive sequence in the corpus was manually classified as follows:

- True Positive — the sequence was found, and its boundaries were identified correctly;
- False Positive — the sequence was found, but extra words were included;
- True Negative — there are no sequences in the sentence, and none were found;
- False Negative — the sequence was found, but some necessary words were excluded.
The precision and recall scores were then calculated, turning out 0.96 and 0.93 respectively.

Then, for each of the sequences found by the program the graphs were drawn to evaluate the kinship relations analysis and visualization. For each of the sequences, the number of expected and present correct visualizations was estimated manually with the resulting accuracy score being 0.95.

B. Discussion

As the evaluation test has outlined flaws in the tool’s performance, a few areas for future work are suggested.

- Processing proper names. As for now, the tool cannot correctly process input sequences including first name + patronym collocations (e.g. сын Ани Ивановны — Anna Ivanovna’s son) or abbreviated name forms (e.g. сын В. Набокова — V. Nabokov’s son).

- Broadening the range of sequence types. For example, the tool cannot correctly process the sequence below because it does not fit any of the sequence type schemas.

  a. шурин моего сына — my wife’s brother’s son

- Adding context analysis features for better differentiation between the sequence types.

- Updating the template approach. At the relations analysis stage, complex kinship terms can be replaced with their simpler explanations, e.g. turning тёща (mother-in-law) into мать жены (a wife’s mother), allowing to only store templates for the basic kinship terms, namely parents, children, siblings and spouses.

- Identifying coreference. At this point, the program does not register several words referring to the same character as in сын моего отца (my dad’s son) and is unable to depict that in the graph.

- Making the tool adjustable for other languages by eradicating the language dependencies in the code.

V. CONCLUSION

This paper has presented kinship term possessive sequences as a field for natural language processing development, presenting the authors’ original tool for the sequences’ human-readable visualization. The program appears quite efficient and performs well on a broad range of input data; however, suggested paths for further development might push its limits significantly.

The project source code is available on github. The evaluation corpus, as well as the list of kinship terms processed by the program, can also be viewed there.

The tool is also available for public use as a Python package. As for now, the users are able to:

- extract sequences from a given piece of text;
- build a graph upon a given sequence;
- visualize an already-built graph or sequences from a given piece of unprocessed text.

REFERENCES
