

SYRCoSE 2018

Editors:

Alexander S. Kamkin, Alexander K. Petrenko, and
Andrey N. Terekhov

Preliminary Proceedings of the 12th Spring/Summer
Young Researchers' Colloquium on Software Engineering

Veliky Novgorod, May 30 – June 1, 2018

Preliminary Proceedings of the 12th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2018), May 30-June 1, 2018 – Veliky Novgorod, Russian Federation.

The issue contains papers accepted for presentation at the 12th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2018) held in Veliky Novgorod, Russian Federation on May 30-June 1, 2018.

The colloquium's topics include programming languages and libraries, software and hardware verification, automata and Petri nets, system programming, and others.

The authors of the selected papers will be invited to participate in a special issue of '*The Proceedings of ISP RAS*' (<http://www.ispras.ru/proceedings/>), a peer-reviewed journal included into the list of periodicals recommended for publishing doctoral research results by the Higher Attestation Commission of the Ministry of Science and Education of the Russian Federation.

The event is sponsored by Russian Foundation for Basic Research (Project №18-07-20034).

Contents

| | |
|---|-----|
| Foreword | 5 |
| Committees | 6 |
| Combining ACSL Specifications and Machine Code <i>P. Putro</i> | 8 |
| Registration Protocol Security Analysis of the Electronic Voting System Based on Blinded Intermediaries using the Avispa Tool <i>I. Pisarev, L. Babenko</i> | 14 |
| Towards Formal Verification of Cyber Security Standards <i>T. Kulik, P.G. Larsen</i> | 20 |
| On the Model Checking of Finite State Transducers over Semigroups <i>A. Gnatenko, V. Zakharov</i> | 26 |
| Tolerant Parsing with a Special Kind of "Any" Symbol: The Algorithm and Practical Application <i>A. Goloveshkin, S. Mikhalkovich</i> | 35 |
| Heterogeneous Architectures Programming Library <i>G. Kirgizov, Ia. Kirilenko</i> | 46 |
| Applying Deep Learning to C# Call Sequence Synthesis <i>A. Chebykin, Ia. Kirilenko</i> | 54 |
| In-Kernel Memory-Mapped I/O Device Emulation <i>V. Cheptsov, A. Khoroshilov</i> | 64 |
| Asymmetric Multiprocessor Problems of Real-Time OS <i>A. Emelenko, A. Tsyvarev, N. Pakulin</i> | 70 |
| Building Modular Real-Time Software from Unified Components Model <i>K. Mallachiev, A. Khoroshilov</i> | 74 |
| Static Verification for Memory Safety of Linux Kernel Drivers <i>A. Vasilev</i> | 80 |
| Configurable System Call Tracer in Qemu Emulator <i>A. Ivanov</i> | 87 |
| Stealth Debugging of Programs in Qemu Emulator with WinDbg Debugger <i>M. Abakumov</i> | 89 |
| An Interactive Specializer Based on Partial Evaluation for a Java Subset <i>I. Adamovich, A. Klimov</i> | 91 |
| Static Dependency Analysis for Incremental Validation of Semantically Complex Data <i>D. Ilyin, N. Fokina, V. Semenov</i> | 97 |
| Source Code Augmentation for Supervised Learning <i>V. Savchenko, A. Volkov</i> | 103 |
| Buffer Overflow Detection via Static Analysis: Expectations vs. Reality <i>I. Dudina</i> | 107 |

| | |
|---|-----|
| An Approach to Simulation-Based Verification of SoC Bus Controllers <i>M. Chupilko, E. Drozdova</i> | 111 |
| Verification of System on Chip Integrated Communication Controllers <i>M. Petrochenkov, R. Mushtakov, D. Shpagilev</i> | 115 |
| Construction of Validation Modules Based on Reference Functional Models in a Standalone Verification of Communication Subsystem <i>D. Lebedev, I. Stotland</i> | 120 |
| Medical Images Segmentation Operations <i>S. Musatian, A. Lomakin, S. Sartasov, L. Popyvanov, I. Monakhov, A. Chizhova</i> | 125 |
| Automatic Detection of Physiologically Singular Points of the Bony Orbit <i>M. Platonova, S. Sartasov, I. Monakhov</i> | 130 |
| The Variants of Chinese Postman Problems and Way of Solving through Transformation into Vehicle Routing Problems <i>M. Gordenko, S. Avdoshin</i> | 133 |
| Applying the Methods of System Analysis to Teaching Assistants' Evaluation <i>E. Beresneva, M. Gordenko</i> | 138 |
| Analysis of Mathematical Formulations of Capacitated Vehicle Routing Problem and Methods for their Solution <i>E. Beresneva, S. Avdoshin</i> | 147 |
| Auto-Calibration and Synchronization of Camera and MEMS-Sensors <i>A. Polyakov, A. Kornilova, Ia. Kirilenko</i> | 153 |
| An Approach to a Software Implementation of Architectural Generative Design for BIM <i>M. Prokofyev, V. Kirillov</i> | 159 |
| Product Reviews Sentiment Analysis in Russian <i>S. Smetanin, M. Komarov</i> | 164 |
| On the Verification of Strictly Deterministic Behaviour of Timed Finite State Machines <i>E. Vinarskii, V. Zakharov</i> | 169 |
| Deriving Adaptive Distinguishing Sequences for Finite State Machines <i>A. Tvardovskii, N. Yevtushenko</i> | 176 |
| Prosega/CPN: An Extension of CPN Tools for Automata-based Analysis and System Verification <i>J.C. Carrasquel, M.E. Villapol, A. Morales</i> | 182 |
| Simulating Behavior of Multi-Agent Systems with Acyclic Interactions of Agents <i>R. Nesterov, A. Mitsyuk, I. Lomazova</i> | 192 |
| Human Readable Extended Finite State Machine Format <i>A. Nikitin</i> | 198 |
| Criteria for Software to Safety-Critical Complex Certifiable Systems Development <i>N. Gorelits, A. Gukova, E. Peskov</i> | 202 |
| Formalizing Metamodel of Requirement Management System <i>D. Kildishev, A. Khoroshilov</i> | 209 |
| Extracting Architectural Information from Source Code of ARINC 653-Compatible Application Software Using CEGAR-Based Approach <i>S. Lesovoy</i> | 215 |

Foreword

Dear participants,

It is our pleasure to meet you at the 12th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). This year's colloquium is hosted by Yaroslav-the-Wise Novgorod State University (NovSU), the largest higher education institution in the Novgorod region. The event is organized by Ivannikov Institute for System Programming of the Russian Academy of Sciences (ISP RAS), Saint-Petersburg State University (SPbSU), and NovSU.

SYRCoSE 2018's Program Committee (consisting of more than 50 members from more than 25 organizations) has selected 36 papers. Each submitted paper has been reviewed independently by two-three referees. The authors and speakers represent well-known universities, research institutes and companies including Aarhus University, Auckland University of Technology, Baumann Moscow State Technical University, Central University of Venezuela, EC-leasing, GosNIAS, Higher School of Economics, ISP RAS, KCD LLC, Keldysh Institute of Applied Mathematics of RAS, Lomonosov Moscow State University, MCST, NovSU, Perm State University, Program Systems Institute of RAS, Sapienza University of Rome, Southern Federal University, SPbSU, Tomsk State University, Università degli Studi di Milano-Bicocca, and Université de Bretagne Occidentale (6 countries, 14 cities, and 23 organizations).

We would like to thank all of the participants of SYRCoSE 2018 and their advisors for interesting papers. We are also very grateful to the PC members and the external referees for their hard work on reviewing the papers and selecting the program. Our thanks go to the invited speakers, Frank Singhoff (Université de Bretagne Occidentale) and Boris Pozin (EC-leasing, Higher School of Economics). We would also like to thank our sponsors: Russian Foundation for Basic Research (Project №18-07-20034) and Exactpro Systems. Our special thanks go to the local organizers, Alexander Cherepitsa, Vladimir Makarov, and Galina Voloshina, for their invaluable help in organizing the colloquium in Veliky Novgorod.


Sincerely yours,

Alexander S. Kamkin
Alexander K. Petrenko
Andrey N. Terekhov

May 2018

Committees

Program Committee Chairs

 Alexander K. Petrenko – Russia
Ivannikov Institute for System Programming of RAS

 Andrey N. Terekhov – Russia
Saint-Petersburg State University

Program Committee

 Elena Yu. Avdieva – Russia
Omsk State Technical University

 Sergey M. Avdoshin – Russia
Higher School of Economics

 Nadezhda F. Bahareva – Russia
Povolzhskiy State University of Telecommunications and Informatics

 Andrey A. Belevantsev – Russia
Ivannikov Institute for System Programming of RAS

 Svetlana I. Chuprina – Russia
Perm State National Research University

 Pavel D. Drobintsev – Russia
Saint-Petersburg State Polytechnic University

 Liliya Yu. Emaletdinova – Russia
Kazan National Research Technical University

 Victor P. Gergel – Russia
Lobachevsky State University of Nizhny Novgorod

 Susanne Graf – France
VERIMAG Laboratory

 Efim M. Grinkrug – Russia
Higher School of Economics

 Maxim L. Gromov – Russia
Tomsk State University

 Vladimir I. Hahanov – Ukraine
Kharkov National University of Radioelectronics

 Shihong Huang – USA
Florida Atlantic University

 Iosif L. Itkin – Russia
Exactpro Systems

 Alexander S. Kamkin – Russia
Ivannikov Institute for System Programming of RAS

 Andrei V. Klimov – Russia
Keldysh Institute of Applied Mathematics of RAS

 Vsevolod P. Kotlyarov – Russia
Saint-Petersburg State Polytechnic University

 Alexander N. Kovartsev – Russia
Samara State Aerospace University

 Vladimir P. Kozyrev – Russia
National Research Nuclear University "MEPhI"

 Daniel S. Kurushin – Russia
State National Research Polytechnic University of Perm

 Peter G. Larsen – Denmark
Aarhus University

 Roustam H. Latypov – Russia
Kazan Federal University

 Alexander A. Letichevsky – Ukraine
Glushkov Institute of Cybernetics, NAS

 Nataliya I. Limanova – Russia
Povolzhskiy State University of Telecommunications and Informatics

 Alexander V. Lipanov – Ukraine
Kharkov National University of Radioelectronics

 Irina A. Lomazova – Russia
Higher School of Economics

 Lyudmila N. Lyadova – Russia
Higher School of Economics

 Vladimir A. Makarov – Russia
Yaroslav-the-Wise Novgorod State University

 Victor M. Malysenko – Russia
Moscow State University

 Tiziana Margaria – Ireland
Lero – The Irish Software Research Centre

 Manuel Mazzara – Russia
Innopolis University

 Alexander S. Mikhaylov – Russia
RN-Form

 Igor A. Minakov – Russia
Institute for the Control of Complex Systems of RAS

 Alexey M. Namestnikov – Russia
Ulyanovsk State Technical University

 Yaroslav R. Nedumov – Russia
Ivannikov Institute for System Programming of RAS

 Valery A. Nepomniaschy – Russia
Ershov Institute of Informatics Systems of SB of RAS

 Mykola S. Nikitchenko – Ukraine
Kyiv National Taras Shevchenko University

 Sergey P. Orlov – Russia
Samara State Technical University

 Elena A. Pavlova – Russia
Microsoft

 Ivan I. Piletski – Belorussia
Belarusian State University of Informatics and Radioelectronics

 Vladimir Yu. Popov – Russia
Ural Federal University

 Yury I. Rogozov – Russia
Taganrog Institute of Technology, Southern Federal University

 Rustam A. Sabitov – Russia
Kazan National Research Technical University

 Nikolay V. Shilov – Russia
A.P. Ershov Institute of Informatics Systems of RAS

 Alberto Sillitti – Russia
Innopolis University

 Ruslan L. Smelyansky – Russia
Moscow State University

 Valeriy A. Sokolov – Russia
Yaroslav Demidov State University

 Petr I. Sosnin – Russia
Ulyanovsk State Technical University

 Veniamin N. Tarasov – Russia
Povolzhskiy State University of Telecommunications and Informatics

 Andrei N. Tiugashev – Russia
Samara State Aerospace University

 Sergey M. Ustinov – Russia
Saint-Petersburg State Polytechnic University

 Vladimir V. Voevodin – Russia
Research Computing Center of Moscow State University

 Dmitry Yu. Volkanov – Russia
Moscow State University

 Mikhail V. Volkov – Russia
Ural Federal University

 Nadezhda G. Yarushkina – Russia
Ulyanovsk State Technical University

 Rostislav Yavorsky – Russia
Higher School of Economics

 Nina V. Yevtushenko – Russia
Ivannikov Institute for System Programming of RAS

 Vladimir A. Zakharov – Russia
Moscow State University

 Sergey S. Zaydullin – Russia
Kazan National Research Technical University

Organizing Committee

Alexander O. Cherepitsa
Yaroslav-the-Wise Novgorod State University

Alexander S. Kamkin
Ivannikov Institute for System Programming of RAS

Vladimir A. Makarov
Yaroslav-the-Wise Novgorod State University

Alexander K. Petrenko
Ivannikov Institute for System Programming of RAS

Galina V. Voloshina
Yaroslav-the-Wise Novgorod State University

Referees

Alhejab Alhazmi

Ivan Andrianov

Nadezhda Bahareva

Andrey Belevantsev

Mikhail Chupilko

Misha Drobyshevskii

Natalia Garanina

Victor Gergel

Aritra Ghosh

Andrey Gomzin

Susanne Graf

Efim Grinkrug

Maxim Gromov

Alexander Kamkin

Andrei Klimov

Vladimir Kozyrev

Mikhail Lebedev

Irina Lomazova

Vladimir Makarov

Victor Malyshko

Manuel Mazzara

Alexander Mikhaylov

Yaroslav Nedumov

Valery Nepomniaschy

Mykola Nikitchenko

Sergey Orlov

Alexander Petrenko

Nikolay Shilov

Alberto Sillitti

Sergey Smolov

Valeriy Sokolov

Petr Sosnin

Andrei Tatarnikov

Andrey Terekhov

Andrei Tyugashev

Andrey Ustyuzhanin

Dmitry Volkanov

Mikhail Volkov

Nina Yevtushenko

Vladimir Zakharov

Combining ACSL Specifications and Machine Code

Pavel Putro

Software Engineering Department

Ivannikov Institute for System Programming of the RAS

Moscow, Russia

pavel.putro@ispras.ru

Abstract—When developing programs in high-level languages, developers have to make assumptions about the correctness of the compiler. However, this may be unacceptable for critical systems. As long as there are no full-fledged formally verified compilers, the author proposes to solve this problem by proving the correctness of the generated machine code by deductive verification. To achieve this goal, it is required to combine the pre- and postcondition specifications with the machine code behavior model. The paper presents an approach how to combine them for the case of C functions without loops.

Index Terms—deductive verification, formal methods, machine code, ACSL

I. INTRODUCTION

The paper presents a step forward towards the creation of a tool capable of proving the correctness of machine code based on the formal specification of a function for a high-level language [1]. Such a tool will allow to avoid the assumption about the correctness of the compiler by verification of the generated code regarding specification of source code functionality. The only way in which the correctness analysis of machine code is not necessary is to create a fully formally verified compiler [2]. However, the existing developments in the field of formally verified compilers [3], at the moment do not allow using all the possibilities of existing unverified analogs, for example, GCC [4]. This work is necessary for the implementation of an alternative approach—deductive verification [5] of compiler products, the correctness of which has not been proven. Using this approach will allow you to safely use the already created software. Different approaches to formal specification and building a model of machine code behavior were proposed in different machine code verification projects. Here, the formal specification of a function or a sequence of machine code instructions shows the pre- and postconditions for a function and the behavior model describes mathematical and logical state change formulas. The paper discusses an approach to combining ACSL [6] specifications of the C language with the machine code of the PowerPC e500mc processor obtained by compiling these functions. The choice of the target language is caused by the fact that most high-critical system software like operating system kernels is written in C. While the very high-level languages support a variety of protective mechanisms such as the prohibition of pointers or checks when casting, the C language is designed for maximum performance by allowing the programmer to interact directly with the memory. Proof of critical code sections by deductive verification methods can

improve the reliability of such systems. In the pursuit of performance, compilers try to make the most of the capabilities of the target processor. Machine code produced by compilers can be extremely difficult for manual verification and specification because the compilation disappears all the information about the names of variables and even the order of execution of commands may be different than in the original program. Only the pre- and postconditions for a particular function remain unchanged. Automatic combination of C-level specifications with the logical model of machine code will allow you to check its correctness in a fully automatic mode.

II. MACHINE CODE REPRESENTATION

The specification of machine code instructions in logical languages is a complex and lengthy process. Often, the appearance of the function behavior model specification in this language is very different from that provided in the processor specification. In addition, the lack of special tools makes it difficult to debug such models. To solve these problems, the author proposes to use the NML language, together with the MicroTESK tool [7]. The NML language contains special structures and data types to simplify the modeling of the hardware. The MicroTESK toolset includes universal disassembler of the machine code by the NML language and the NML to SMT-LIB [8] translator. “Fig. 1” shows the `cmpl` operation specification from the official documentation for PowerPC e500 core family [9] processors and “Fig. 2” shows its NML version. From here, you can see that the NML language allows you to fully describe processor instructions, including their representation in Assembly language and machine code. In addition, the use of the NML language as the basis for the representation of machine code will allow to reuse all NML models, developed by the MicroTESK development team for the purposes of testing of microprocessors.

III. ACSL SPECIFICATIONS REPRESENTATION

A. ACSL specifications translation

As a logical language in which ACSL specifications will be translated, the author suggests using the WhyML language [10]. The Why3 tool, designed to analyze this language, allows you to apply many useful transformations and optimizations. It also allows you to translate WhyML code into logical code for many different provers. In addition, the task of translating ACSL specifications into WhyML code has already been solved by the Jessie plugin [11] for Frama-C [12]. In the

Compare Logical

cmpl **crfD,L,rA,rB**

| | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---------------|---|---|---|---|-------|---|---|----|--|-------|----|--|--|---|-------|---|---|---|---|---|---|---|---|---|
| 0 | 5 6 8 9 10 11 | | | | | 15 16 | | | | | 20 21 | | | | | 30 31 | | | | | | | | | |
| 0 | 1 | 1 | 1 | 1 | 1 | crfD | / | L | rA | | | rB | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | / |

```

if L=0 then a ← 320 || (rA)32:63
else      a ← (rA)
if L=0 then b ← 320 || (rB)32:63
else      b ← (rB)
if a <u b then c ← 0b100
if a >u b then c ← 0b010
if a = b then c ← 0b001
CR4×crfD+32:4×crfD+35 ← c || SO

```

Fig. 1. CMPL official specification.

```

op cmpl (crfD: CRFD, L: BIT, ra: R, rb: R)
  init = {
    XO_10 = coerce(card(10), 0b000001000000);
    OPCD = coerce(card(6), 0b0111111);
  }
  syntax = format("cmpl %d, %d, %s, %s", crfD, L, ra.syntax, rb.syntax)
  image = format("%6s%3s%1s%1s%5s%5s%10s%1s", OPCD, crfD, L, "0", ra.image, rb.image, XO_10,
  action = {
    if L == coerce(BIT, 0) then
      if ra < rb then
        temp = 0b001;
      endif;
      if ra > rb then
        temp = 0b010;
      endif;
      if ra == rb then
        temp = 0b100;
      endif;
      CR<(coerce(card(5), crfD)*4+2) .. (coerce(card(5), crfD)*4)> = coerce(card(3), temp);
      CR<coerce(card(5), crfD)*4+3> = XER_SO;
    endif;
  }
}

```

Fig. 2. CMPL NML specification.

course of research [1], it was established that the use of the plugin Jessie directly, not suitable for the tasks of machine code analysis. Jessie plugin makes a number of simplifying assumptions that do not take into account the peculiarities of machine code. Instead, it was decided to take as a basis the unfinished code of jessie3 project [13] part of the Why3 project. The Jessie3 code has been modified and extended to take into account the peculiarities of machine code. In particular, the language WhyML has been described the type

of processor registers. In addition, the algorithm of generating targets for the proof was changed for the subsequent fusion pre- and postconditions were separated from the function behavior model.

B. Using register type for compatibility with machine code

Processor registers can be represented by a limited integer type with an extended set of operations. Operations include signed and unsigned arithmetic, bitwise operations, and memory read operations at the address specified in the register and

by offset. To describe all such operations high-level languages, use a variety of different types, as well as a cast operation. However, using different data types will complicate the proof of correctness problem for SMT-solvers. This is especially noticeable in the case of bitwise operations, which are available only for bitvectors in SMT-LIB. Bitvectors cast operations to an integer type are not supported by the latest SMT-LIB [14] standard, and various SMT-solvers offer their own version of the implementation of this operation. The BitVec type from SMT-LIB is well-suited for describing the type of registers because it contains all the necessary arithmetic and logical sign and unsigned operations. However, the theory of bitvectors at the why3 level does not support all the necessary operations and is built as an unsigned type. Based on the standard theory of bitvectors, the author developed a theory to support the type of processor registers. The theory supports both signed and unsigned integer types and there is ongoing work to add support for pointer arithmetics and memory dereferencing. The driver for CVC4 SMT-solver [15] was updated for translation of the register type to the type BitVec with corresponding mapping of operations.

C. Splitting specification and behavior model

To merge machine code, you must separate the pre- and post-conditions from the behavior of the high-level function, which will then be replaced by the behavior of the machine code. To implement this approach, the author uses abstract logical predicates of pre- and postconditions checking. These predicates take as input the parameters of the verification function, and the predicate of the postcondition is also taking its result. Further, by means of axioms predicates are defined by a logical expression in accordance with ACSL specifications. In “Fig. 3” you can see the predicates for pre- and postconditions are generated based on the ACSL specifications of absolute value function (“Fig. 4”), where `usabs_pre` the predicate of a precondition, and `usabs_post` is a predicate of the postcondition.

D. Replacing proof goal

To facilitate the subsequent merging, the proof goal is substituted during translation of WhyML to SMT-LIB. A new goal for the proof can be described as follows: If the precondition of a function with its arguments is satisfied then the postcondition with the arguments of the function and its result is not satisfied. The negation is used because the SMT-solvers operation specifics searching for example variable values that will satisfy all restrictions described in SMT-LIB model. If such an example could not be found, then the assumption is incorrect and the predicate of the postcondition is always executed. Therefore, the Expected verdict of the SMT-solver `unsat`. It is important to note here that arguments and the result of the function execution are not associated with machine code at this stage the merge module solves the problem of their binding. “Fig. 5” shows SMT-LIB code of goal to prove the correctness of the absolute value function.

IV. MERGING HIGH- AND LOW-LEVEL SPECIFICATIONS

If you perform all the steps described in the previous sections of this paper, namely, creating an NML model of the machine code and an ACSL to the WhyML translation module, you can perform a merge in two different ways. The first method is the merging at the level of WhyML, and the subsequent translation to SMT-LIB by means of Why3. This approach has a number of advantages, mainly related to Why3 capabilities for WhyML code analysis. It is worth noting that Why3 IDE (“Fig. 6”), can be used for interactive proof and manual simplifications of verification goals. At the moment the MicroTESK team, with the support of the author, is developing an NML to WhyML translation module. The second approach, as well as the only one implemented at the moment, is merging at the SMT-LIB level. The main advantage of this approach is that the MicroTESK tool has already been implemented NML to SMT-LIB translation module. In addition, the vast majority of operations and data types available in NML have analogs in SMT-LIB. For example, a set of General-purpose registers is modeled in the NML of the PowerPC processor model as an array of 32-bit registers with a 5-bit index. There is no predefined 5-bit unsigned type in Why3, let alone an array with such an index. However, in SMT-LIB, as in NML, you can manually set the length of BitVec constants. In addition, the translation directly to SMT-LIB allows to avoid unnecessary abstractions that Why3 algorithm for WhyML to SMT-LIB translation can add. The task of the merge module is to bind together the function arguments and the result of function of high-level language with registers and memory of the model of machine code, and set the environment. Here, the environment refers to machine-specific things, such as the initial value of the stack register or instruction counter. To do this, it is necessary to take into account the specificity of generation SMT-LIB behaviors of the machine code and the specification for the function and specificity of the ABI of architecture. Next, in “Fig. 7” we can see binding of the arguments of instructions with the registers for the PowerPC architecture. Developed by the MicroTESK team, generation SMT-LIB by the NML model produces thousands of lines of code. This code can be divided into two main parts: The declaration of all the logical constants needed to describe the behavior model and the description of the state transformation formulas by means of using one `assert` per machine code instruction and one for every of machine instruction argument.

V. EVALUATION

The developed approach was successfully used to verify the machine code of the absolute value function on the basis of bitwise operations (“Fig. 8”), for which a verdict was obtained, clearly indicating correctness of the function. Tests were also developed to verify the correctness of the implementation of translation of mathematical and logical operations of the ACSL language. Testing of the NML model was done by means of MicroTESK tool.

```

predicate usabs_post r32 r32

predicate usabs_pre r32

axiom usabs_post_axiom :
  forall n:r32, result:r32.
  usabs_post n result <=>
    sge result (of_int 0) /\ (eq result n \/ eq result (sub (of_int 0) n))

axiom usabs_pre_axiom :
  forall n:r32. usabs_pre n <=> slt (neg (of_int 2147483648)) n

```

Fig. 3. WhyML abs specification.

```

/*@ requires -2147483648 < n;
    ensures \result == n || \result == 0-n;
    ensures \result >= 0;
*/
int abs(int n)

```

Fig. 4. ACSL abs specification.

```

;;function argument 1
(declare-const _arg (_ BitVec 32))
;;assign _arg here

;;function result
(declare-const _func_res (_ BitVec 32))
;;assign _func_res here

(assert (usabs_pre
_arg ))

(assert (not (usabs_post
_arg _func_res)))

```

Fig. 5. Proof goal template.

VI. RELATED WORKS

In the why3-avr [16] [17] project, the deductive verification approach is used to prove the correctness of non-loop programs in the assembly language of the AVR microcontroller. The AVR microcontroller used in this study has a fairly simple instruction set that allows you to manually specify the behavior model for each command in the WhyML language, which does not have special means to describe such structures. Also, the model code is described in such a way that allows the programmer to simply copy the function code in the AVR assembly language and add to it a formal specification to get WhyML code for checking the correctness of the function. This approach is especially useful for direct development in a low-level language because the Why3 tool has rich capabilities for transformation and analysis of Why3 code. Also, the use of Why3 allows converting the WhyML code for proving by various SMT solvers. However, the program in assembly

language is different from compiled machine code that in machine code is a sequence of bytes where there is no all information associated with label names and variables, as well as the formal specification. In addition, machine code does not allow you to abstract from your environment as much as assembly language code. For example, in machine code, indicators such as the address of a function in memory and the value of the stack register at the time of entering the function are important. Also, a high-level formal language specification, such as C, uses various abstractions, such as parameter names and variables, that become unavailable after they are translated into assembly language or machine code. The approaches proposed by the author differ from those described in this project in that they allow using the specification of the high-level language function for analyzing machine code, as well as scaling the supported command system with the help of a specialized modeling language hardware NML.

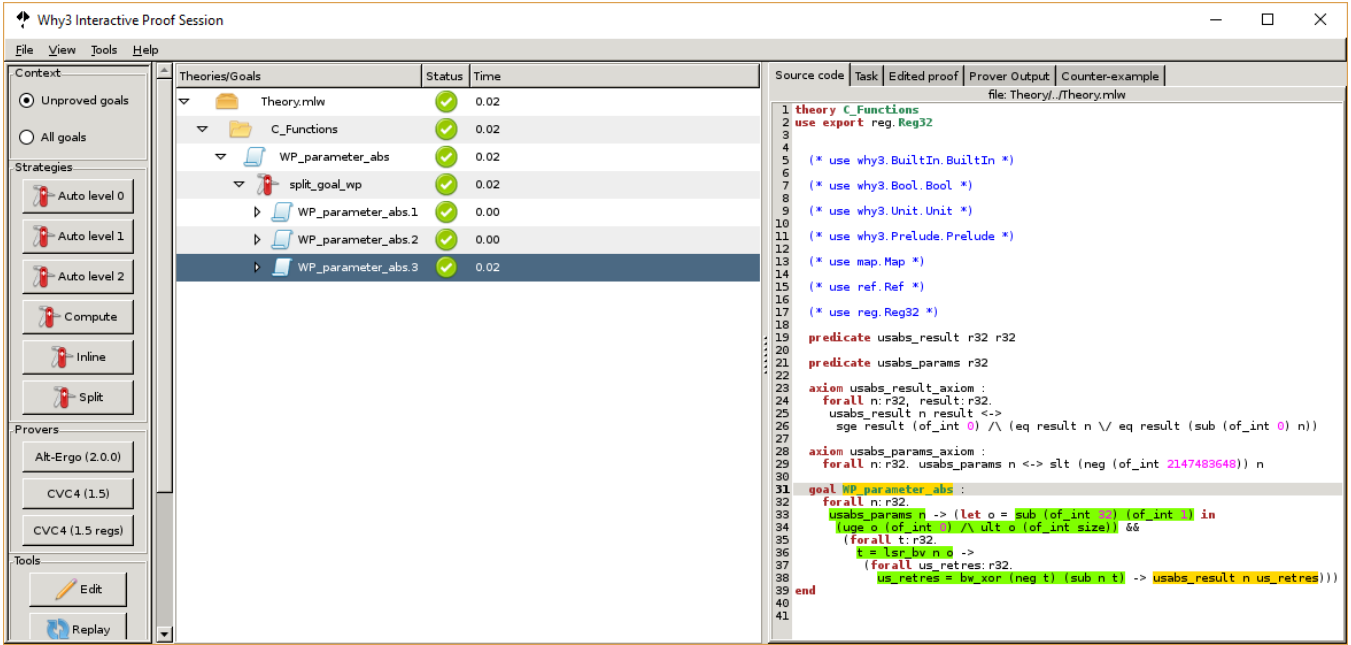


Fig. 6. Why3 IDE.

```
;;function argument 1
(declare-const _arg_1 (_ BitVec 32))
(assert (= _arg_1 (select GPR!1 (_ bv3 5))))

;;function result
(declare-const _func_res (_ BitVec 32))
(assert (= _func_res (select GPR!47 (_ bv3 5))))
```

Fig. 7. Binding function argument and result.

```
int abs(int n)
{
    int t = (unsigned int) n >> (32-1);
    return (-t) ^ (n-t);
}
```

Fig. 8. Absolute value function.

In the Technical report published by the University of Cambridge Computer Laboratory [18] the HOL4 proof assistant [19] is used for Formal verification of machine-code programs. The paper describes a tool able to verify the machine code for subsets of instructions for popular architectures ARMV4, PowerPC, x86. Behavior model for these instructions was developed by independent developers, so models for both ARM and x86 was designed for HOL4 language [20] [21], and the PowerPC model [22] were manually translated from the Coq language [23] to HOL4. Here it is worth noting the similarity with the project why3-avr because instructions behavior models were specified manually on unspecialized for such a purpose language. The report terminology uses four levels of abstraction to describe the logical implementation

and specification of functions. To obtain a low-level function model (level 2) automatic decompiler translates the machine code (level 1) into recursive functions on the HOL4 language, and also generates their specifications. The use of recursion, in this case, avoids the need to define loop invariants. The user can then focus on interactively proving the properties of the generated function using the HOL4 proof assistant. For verification, the user also needs to describe the high-level model of the function (level 3), as well as the specification of the function for (level 4). Further, by using relations between levels, user proves that the machine code model complies with the functional specification. In contrast to the interactive HOL4 approach, the approach used in the author's study allows the presence of ACSL specifications to carry out all stages in

automatic mode. Also in the author's approach to proving the correctness of machine code is not necessary to have a logical model of the behavior of the function in a high-level language. This degree of automation is achieved including the use of automatic SMT-solvers, in contrast to the interactive proof assistant HOL4. Particularly worth noting is the approach to the translation of programs into recursive functions. The use of high-level language loop invariants at the machine code level is extremely difficult due to the influence of various compiler optimizations. The recursive functions may help to solve this problems.

A number of papers also describe the use of model checking [24] approach for formal verification of machine code. Therefore, in the paper [25] for verification of machine code of the microcontroller Motorola M68hc11 is used Bogor framework [26]. This approach does not imply the presence of function contracts but is based on the use of formally specified behavior models of the system as a whole. As a result, it can be said that the scope of the requirements to be tested varies with the use of deductive verification and model checking.

VII. CONCLUSION

Most of the work that is reviewed specifies the behavior of machine code instructions manually in the logical language. However, in order to simplify and improve the reliability of processor models, the author proposed to describe them in the NML language, designed specifically for such purposes, with the subsequent automatic translation of the model into logical languages. The use of this approach is also facilitated by the presence of a large set of tools in the MicroTESK tool to work with NML, including the NML to SMT-LIB translator. The particularity of ACSL specifications translation to WhyML code, for the case of verification of machine code, such as the need to separate the specification from the behavior model, as well as the importance of the introduction and implementation of the register type. The observance of such rules and guidelines will allow for automatic merging of function specification and machine code behavior model and thus avoid the need for manual specifying machine code behavior model on the logical language, as required in the project why3-avr. There were proposed two approaches to merge of code specifications and behavior models: at the level of WhyML, and at the level of the SMT-LIB. The first approach allows to use SMT-LIB code generated directly from NML model that help us to avoid extra complexity coming from double translation NML to WhyML and then WhyML to SMT-LIB. The second approach allows to use all the features of the Why3 tool, such as interactive transformations and support of various provers and solvers.

The use of the methods and approaches described in this paper will allow you to fully automate deductive verification of machine code without loops for compliance with the contract specification in ACSL language.

REFERENCES

[1] Putro P. An Investigation into the Possibility of Analyzing Binary Code with SMT Solvers. HSE Moscow 2017

[2] Leroy, Xavier (2009-12-01). A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*. 43 (4): 363. doi:10.1007/s10817-009-9155-4. ISSN 0168-7433.

[3] "CompCert - The CompCert C compiler". *comp-cert.inria.fr*. Retrieved 2018-13-02.

[4] "GCC Releases". <http://www.gnu.org/software/gcc/releases.html> GNU Project. Retrieved 2018-13-02.

[5] Butterfield A., Ngondi G., Kerr A. "A Dictionary of Computer Science" (ed. 7)

[6] "ACSL specification". <http://frama-c.com/acsl.html> Retrieved 2018-13-02

[7] Kamkin A., Tatarnikov A. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. *Proceedings of the 6th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE)*.

[8] C Barrett, R Sebastiani, S Seshia, and C Tinelli, *Satisfiability Modulo Theories*. In *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, (A Biere, M J H Heule, H van Maaren, and T Walsh, eds.), IOS Press, Feb. 2009, pp. 825885.

[9] EREF: A Programmers Reference Manual for Freescale Power Architecture Processors, Rev. 1 (EIS 2.1)

[10] Fillitre JC., Paskevich A. (2013) Why3 Where Programs Meet Provers. In: Felleisen M., Gardner P. (eds) *Programming Languages and Systems. ESOP 2013*. *Lecture Notes in Computer Science*, vol 7792. Springer, Berlin, Heidelberg.

[11] M. Mandrykin, A. Khoroshilov "A Memory Model for Deductively Verifying Linux Kernel Modules". In *Lecture Notes in Computer Sciences #10742 "Perspectives of System Informatics: 11th International Andrei P. Ershov Informatics Conference"*, pp. 256-275, Springer International Publishing. DOI: 10.1007/978-3-319-74313-4_19

[12] Frama-c: A Software Analysis Perspective /Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov et al. // *Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM12*. Berlin, Heidelberg: Springer-Verlag, 2012. Pp. 233247. http://dx.doi.org/10.1007/978-3-642-33826-7_16.

[13] Jessie3 at Why3 source repository. <https://gitlab.inria.fr/why3/why3/tree/master/src/jessie>

[14] arett C., Fontaine P., Tinelli C. *The SMT-LIB Standard Version 2.6*. 2017-07-18

[15] Barrett C. et al. (2011) CVC4. In: Gopalakrishnan G., Qadeer S. (eds) *Computer Aided Verification. CAV 2011*. *Lecture Notes in Computer Science*, vol 6806. Springer, Berlin, Heidelberg

[16] Marc Schoolderman *Verifying Branch-Free Assembly Code in Why3*.

[17] "Why3-avr project repository" <https://gitlab.science.ru.nl/sovereign/why3-avr>

[18] Magnus O. Myreen. *Formal verification of machine-code programs*. University of Cambridge Computer Laboratory 2009

[19] Konrad Slind and Michael Norrish. *A brief overview of HOL4*. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[20] Anthony Fox. *Formal specification and verification of ARM6*. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2003.

[21] Karl Crary and Susmit Sarkar. *Foundational certified code in a meta-logical framework*. Technical Report CMU-CS-03-108, Carnegie Mellon University, 2003.

[22] Xavier Leroy. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*. In *Principles of Programming Languages (POPL)*. ACM, 2006.

[23] Yves Bertot. *A short presentation of Coq*. In *Theorem Proving in Higher Order Logics (TPHOLs)*. Springer, 2008.

[24] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen "Systems and Software Verification: Model-Checking Techniques and Tools"

[25] Joseph R. Edelman *Machine Code Verification Using The Bogor Framework*

[26] "Bogor framework homepage" <http://bogor.projects.cs.ksu.edu> Retrieved 2018-13-02. 2018-13-02.

Registration protocol security analysis of the electronic voting system based on blinded intermediaries using the Avispa tool

1st Ilya Pisarev

Information security department
Southern Federal University
Taganrog, Russian Federation
ilua.pisar@gmail.com

2nd Liudmila Babenko

Information security department
Southern Federal University
Taganrog, Russian Federation
lkbabenko@sfedu.ru

Abstract—Electronic voting systems are a future alternative to traditional methods of voting. It is important to verify the main algorithms on which system security is based. This paper analyzes the security of the cryptographic protocol at the registration stage, which is used in the electronic voting system based on blind intermediaries created by the authors. The registration protocol is described, the messages transmitted between the parties are shown and their content is explained. The Dolev-Yao threat model is used during protocols modeling. The Avispa tool is used for analyzing the security of the selected protocol. The protocol is described in CAS+ and subsequently translated into the HLPSL (High-Level Protocol Specification Language) special language with which Avispa work. The description of the protocol includes roles, data, encryption keys, the order of transmitted messages between parties, parties' knowledge include attacker, the purpose of verification. The verification goals of the cryptographic protocol for resistance to attacks on authentication, secrecy and replay attacks are set. The data that a potential attacker may possess is detected. The security analysis of the registration protocol was made. The analysis showed that the objectives of the audit were put forward. A detailed diagram of the messages transmission and their contents is displayed in the presence of an attacker who performs a MITM-attack (Man in the middle). The effectiveness of protocol protection from the attacker actions is shown.

Keywords—*e-voting, cryptographic protocols, cryptographic security, cryptographic protocols security verification.*

I. INTRODUCTION

The creation of e-voting systems is a serious problem. There are a number of ready-made systems [1,2] that are used in practice, but they are far from a sufficient level of reliability and the presence of necessary mechanisms, such as complete anonymity of the voter or vote checking opportunity after counting stage. There are also a lot of works in which perspective methods of conducting electronic voting are considered, based on such principles as homomorphic encryption, including threshold schemes, mix-net, secret sharing schemes and others [3-16]. However, in most cases, the authors of such works show theoretical calculations, from which the basic structural unit of interaction between parties does not follow, namely, cryptographic protocol. Any method on which electronic voting is based, no matter how good it is, loses its security if there are any flaws in the structure of

cryptographic protocol that lead to various attacks by the intruder. Thus, the goal of this paper is to test the cryptographic protocol in the important registration stage from various attacks, such as: attack on parties' authentication, data privacy and replay-attacks using the Avispa tool [17].

II. AVISPA TOOL

Avispa is a tool for automated security analysis of cryptographic protocols [17]. With the help of Avispa, in the context of the developed protocols, it is possible to verify: the parties' authentication, the secrecy of data and protection against replay-attacks. It is impossible to perform integrity checks, in particular, used in protocol CMAC mode (Cipher-based message authentication code) using the Avispa tool. The protocol does not imply the use of timestamps in their classic implementation as a part of message. Instead, the developed system uses a temporary session control by server, in which long live sessions are broke down.

In the paper registration stage is analyzed. Three sides are modeled: user, server-intermediary and main server. The protocol will be analyzed after the phase of common session key distribution between the parties. The protocol will be described in CAS+ [18] language, then translated using the Avispa translator into HLPSL [19]. The check will be carried out using the On-the-Fly Model Checking (OFMC) module, where the verification goals are the transmitted data confidentiality and parties' authentication.

For verification, it is necessary to describe the protocol in one of the formal languages: CAS+ or HLPSL. The first language is simpler in syntax and allows you to quickly describe the protocol. An example of syntax is shown below:

```
protocol NeedhamSchroederPublicKey;
identifiers
A, B                : user;
Na, Nb              : number;
KPa, KPb            : public_key;

messages
1. A -> B           : {Na, A}KPb
2. B -> A           : {Na, Nb}KPa
3. A -> B           : {Nb}KPb

knowledge
A                   : A, B, KPa, KPb;
```

```

B      : A, B, KPa, KPb;

session_instances
[A:alice, B:bob, KPa:ka, KPb:kb];

```

The second language HPSL is the language with which Avispa works directly. An example of syntax is shown below:

```

role Alice (A, B: agent,
            KPa, KPb: public_key,
            SND, RCV: channel (dy))
  played_by A def=
  transition

0. State = 0 /\ RCV(start) =>
   State' := 2 /\ Na' := new() /\
   SND({Na'.A}_KPb)
   ...

role Bob(A, B: agent,
         KPa, KPb: public_key,
         SND, RCV: channel (dy))
  ...

```

The syntax of this language is more difficult and the best way to describe the protocol is to describe it in CAS+, and then use Avispa to convert it to HPSL. It is worth to say that if the more complex and larger your protocol, then there is greater chance of errors occurring during translation, so after that you need manually to fix some fragments in HPSL. It's also worth to say that you should not describe the goals of checking in CAS +, but rather add them directly in HPSL.

During protocols describing, the following entities are used: roles, data, message order, sessions and verification purposes. After the description of the protocol, including the indication of verification objectives, it is possible to analyze protocol security against attacks. For analyzing, you can use different modes, but the most effective is the OFMC mode (see Figure 1).

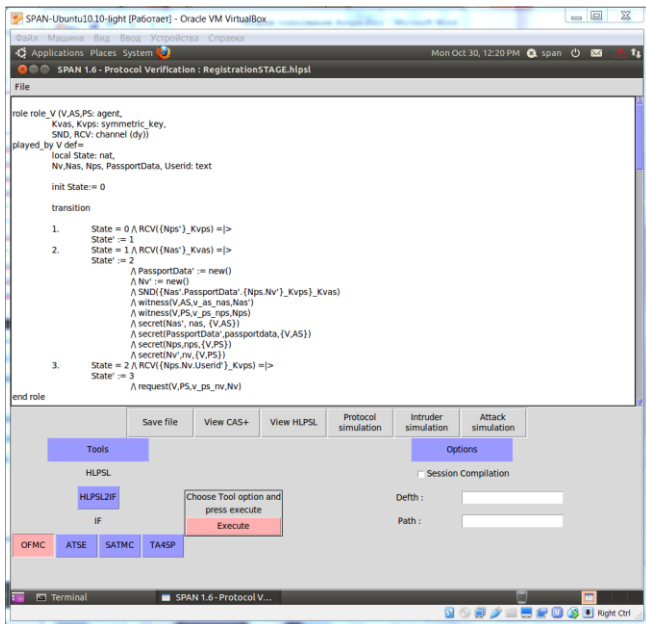


Figure 1 – UI

It requires an additional specification for all data involved in the verification, as well as the message area where verification is required for party authentication. As a result of verification, the corresponding result will be issued. In case of attacks detection, the type of attack and its progress will appear in the form of corresponding changes in messages by the intruder, as in Figure 2.

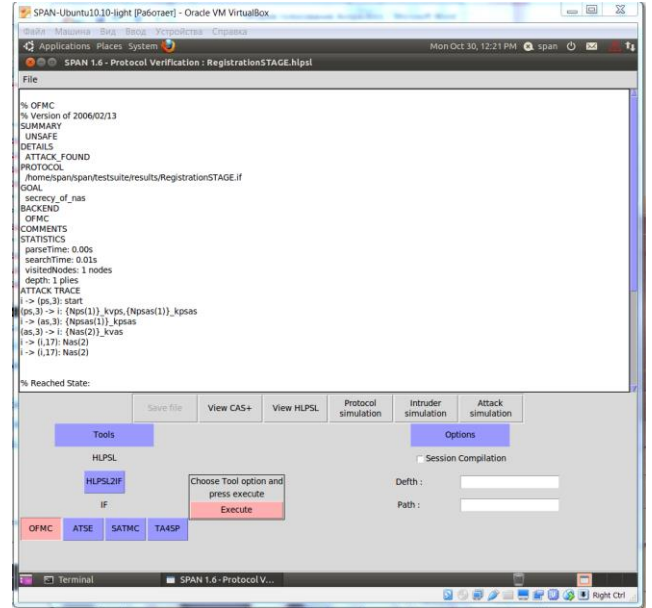


Figure 2 – Founded attack trace

If there are no attacks, then the program output will contain a corresponding message that protocol is safe (see Figure 3).

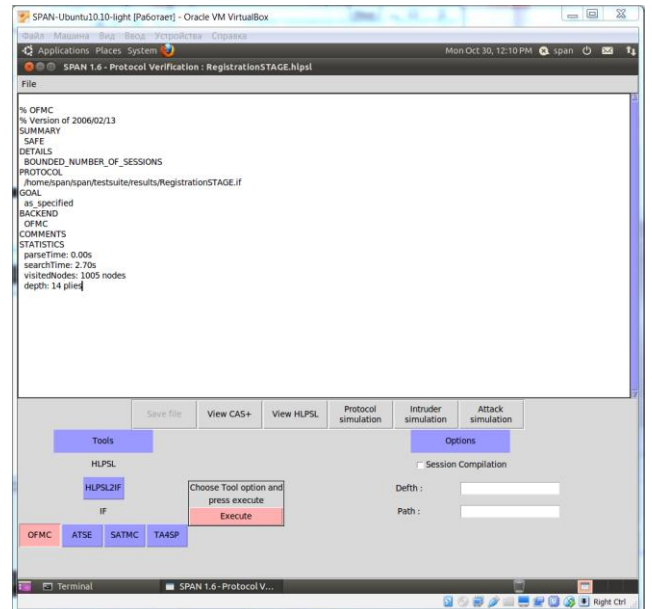


Figure 3 – Result after verification of safe protocol

Using the "Protocol simulation" button, you can see the interaction scheme of the parties in your protocol. With the

help of the button "Intruder simulation" such a scheme will appear, only with the participation of the intruders' side, in which the data intercepted by him will appear. With the help of the button "Attack simulation" you can see the scheme of the attack with intruder, provided that there is an attack in your protocol.

III. E-VOTING SYSTEM DESCRIPTION

A. System architecture

The system architecture is based on the use of the following components: client application for voter - V, 3 server applications that will be located on different physical machines: AS (authentication server), PS (processing server), VS (voting server), encryption application for the passport database and ballots DBE (database encryptor). The general scheme of the interaction of components is shown in Figure 4.

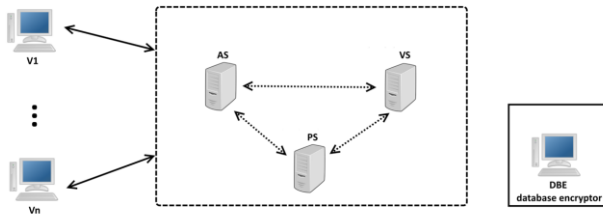


Figure 4 – System architecture

The basic principle on which the system protocols are based - blinded intermediaries (see Figure 5).

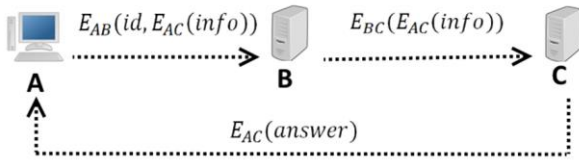


Figure 5 – Blinded intermediaries principle

There are 3 interacting sides A, B, C. Using the protocol for generating a common secret key, the session key AB, BC, AC are generated. A encrypts some information *info* on the AC key, appends an *id* to it, encrypts it on the AB key and sends this message to B. B in this case is a blinded intermediary, because it can decrypt only the first part of the message with *id*, and the remainder with *info* can not. It accepts the message, decrypts and checks if *id* is in the database and, then redirects the remainder of the message encrypted again on the BC key to the C side. C receives the message, decrypts *info*, encrypts the answer response on the AC key and sends it to A. This principle ensures that: *info* will be accepted only if *id* is in the database and that it is impossible to correlate *id* with *info*.

B. Stages description

Stages of electronic voting in the context of the system:

1. Preparation. At this stage, a database of voters and a ballot are created. This data is encrypted, and officials deliver this data to the appropriate server components of the system.

2. Registration. At this stage, users log in to the system using their identification data, at the moment - using passport data, and they get their anonymous identifier. It should be noted that by using the previously described principle of blind intermediaries, it is impossible to correlate open passport data with an anonymous identifier, which ensures the requirement of anonymity.

3. Voting. Users receive a ballot, make their choice and send filled ballot with their anonymous identifier to the server. If such an identifier is present, the vote is accepted, and the verification identifier is sent to user, with which he or she can check vote after counting stage. It is worth noting that it is very important that the user can check his vote after the counting.

4. Counting results and votes checking. At the last stage, the votes are counted, the results are published in the public domain, and any voted user can check his or her vote with a verification identifier.

IV. REGISTRATION STAGE

The electronic voting system based on blind intermediaries, includes a registration stage in which the voter is given anonymous identifier after presenting his passport data. A simplified scheme of the registration stage is shown in Figure 6.

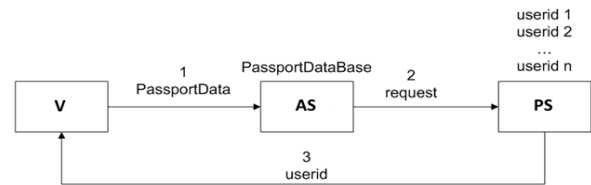


Figure 6 – Simplified scheme of registration stage.

Secret keys V, VAS, VPS are generated using the protocol for generating a common session key. The server parties generate random numbers and send messages (1), (2), (3) to their recipients. They will be used for parties' authentication. V generates N_v . Next, it generates a message (4) with the passport data, which is a hash from a set of document fields, encrypted random numbers on the shared secret key VPS, calculates the CMAC, encrypts all this data on VAS key, calculates the CMAC and sends to AS. AS in this case is a blinded intermediary. It checks the message integrity by CMAC checking, searches *PassportData* in the database and, if successful, redirects another part of the message (5) to side PS. PS checks integrity, if successful, generates *userid*, adds it to database and sends to V as a message (6). The voter decrypts the message, checks integrity and values of random numbers, and remembers his anonymous unique identifier *userid*, with which the user can vote.

```

ECDHE (V, AS) – vas
ECDHE (V, PS) – vps
ECDHE (PS, AS) – psas
V: generates  $N_{as}$ 
(1) AS  $\rightarrow$  V:  $E_{vas}(N_{as})$ 
PS: generates  $N_{ps}$ 
(2) PS  $\rightarrow$  V:  $E_{vps}(N_{ps})$ 
PS: generates  $N_{psas}$ 
(3) PS  $\rightarrow$  AS:  $E_{psas}(N_{psas})$ 
V: generates  $N_v$ 
(4) V  $\rightarrow$  AS:  $E_{vas}(N_{as}, \text{PassportData}, E_{vps}(N_{ps}, N_v), \text{CMAC1}), \text{CMAC2}$ 
AS  $\rightarrow$  V: "Success"
(5) AS  $\rightarrow$  PS:  $E_{psas}(N_{psas}, E_{vps}(N_{ps}, N_v), \text{CMAC1}), \text{CMAC3}$ 
PS: generates userid
(6) PS  $\rightarrow$  V:  $E_{vps}(N_{ps}, N_v, \text{userid}), \text{CMAC4}$ 

```

ECDHE is a Diffie-Hellman protocol on elliptical curves using ephemeral keys. In our case, we use a modified version of ECDHE-RSA, where authentication is done using a signature RSA and a server certificate which help to prevent MITM (man in the middle) attacks. The protocol description is as follows.

```

ECDHE:
(1) V  $\rightarrow$  S: "Hello"
(2) S  $\rightarrow$  V:  $DHs, \text{Sign}_{SKs}(DHs), \text{Certificate}$ 
(3) S: Checks Certificate and  $\text{Sign}_{SKs}(DHs)$ 
(4) V  $\rightarrow$  S:  $DHv$ 
(5) Both sides generate a common session key K for further
interaction with a symmetric cipher.

```

Here V is the client, S is the trusted server that has the certificate, DHs is the server secret part, DHv is the client secret part, $\text{Sign}_{SKs}(DHs)$ is the signature with the server's private key SKs , *Certificate* is the server certificate.

When servers generate common secret key, the same protocol is used, except that both parties exchange certificates and if they are valid, a common session key is generated. The security verification of the registration protocol will be carried out after this stage.

V. SECURITY ANALYSIS OF REGISTRATION PROTOCOL USING AVISPA TOOL

Consider the description of the protocol in CAS + at the registration stage.

```

1  protocol EVotingRegistration;
2  identifiers
3  V,AS,PS: user;
4  Nas,Nps,Npsas,Nv,PassportData,Userid : number;
5  Kvas,Kvps,Kpsas: symmetric_key;
6
7  messages
8  1. PS  $\rightarrow$  V      : {Nps}Kvps
9  2. PS  $\rightarrow$  AS    : {Npsas}Kpsas
10 3. AS  $\rightarrow$  V     : {Nas}Kvas
114. V  $\rightarrow$  AS : {Nas,PassportData,{Nps,Nv}Kvps}Kvas
125. AS  $\rightarrow$  PS : {Npsas,{Nps,Nv}Kvps}Kpsas
136. PS  $\rightarrow$  V   : {Nps,Nv,Userid}Kvps
14
15 knowledge
16 V:V,AS,PS,Nas,Nps,Nv,PassportData,Userid,Kvas,Kvps
17 PS:V,AS,PS,Nps,Npsas,PassportData,Kvas,Kpsas
18 VS:V,AS,PS,Npsas,Nps,Nv,Userid,Kvps,Kpsas
19
20 session_instances

```

```

21 [V:v,AS:as,VS:ps,Kvas:kvas,Kvps:kvps,Kpsas:kpsas]
22 [V:v,AS:as,VS:ps,Kvas:kvas,Kvps:kvps,Kpsas:kpsas];
23
24 intruder_knowledge
25 v,as,ps;
26
27 goal
28 secrecy_of Nps [V,PS];
29 secrecy_of Npsas [AS,PS];
30 secrecy_of Nas [V,AS];
31 secrecy_of Nv [V,PS];
32 secrecy_of PassportData [V,AS];
33 secrecy_of Userid [V,PS];
34 AS authenticates V on Nas;
35 PS authenticates AS on Npsas;
36 PS authenticates V on Nps;
37 V authenticates PS on Nv;

```

Three interacting parties are described as roles: V, AS, PS (lines 2-3). The identifiers section describes the objects participating in the protocol: interacting parties (line 3), random numbers for authentication, identifiers (line 4). Symmetric keys are specified that will be used for message encryption (line 5). The messages section (lines 7-13) describes the transfer of messages between roles, which data is transmitted, and on which key it encrypted. The knowledge section (lines 15-18) describes roles' data knowledge during the execution of the protocol. In the *session_instances* section (lines 20-22), sessions are described. Among the simulated sessions, 2 are allocated, which allow simulating interaction of two clients with the system. This will detect possible attacks on the parties' authentication and replay-attacks. The *intruder_knowledge* section (lines 24-25) specifies the original knowledge of the intruder. In the goal section (lines 27-37) the secrecy of important values is indicated and the authentication according to the request-response scheme with the transfer of random numbers between the participants. For secrecy of the value, it is necessary that this variable is encrypted and that the encryption key does not come to intruder. In order for one party to authenticate another using the request-response mechanism, it is required that the party wanting to authenticate send a random number to the other party, and that other party in the response message returns this random number. In this protocol there are 4 such actions:

1. AS authenticates V by Nas
2. PS authenticates AS by Npsas
3. PS authenticates V by Nps
4. V authenticates the PS to Nv

As for replay-attacks, protection against them is possible due to the presence of a random number at the beginning of each message, which each side checks when message is received. The results of the check using the OFMC module are shown in Figure 7.

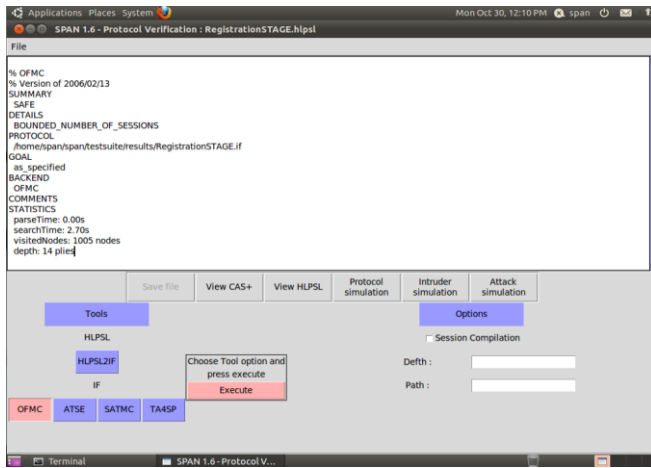


Figure 7 – Registration protocol verification using OFMC mode.

Figure 8 shows the scheme of interaction between the parties at the stage of registration by steps. Figure 9 shows the interaction scheme in the presence of an intruder (*Intruder* side, highlighted in red). This scheme is a visual implementation of the attack man in the middle. When transmitting messages during execution, a transition is made from the "Incoming events" area to "Past events", and the format is the direction of message transfer (from whom and to whom) and the message itself.

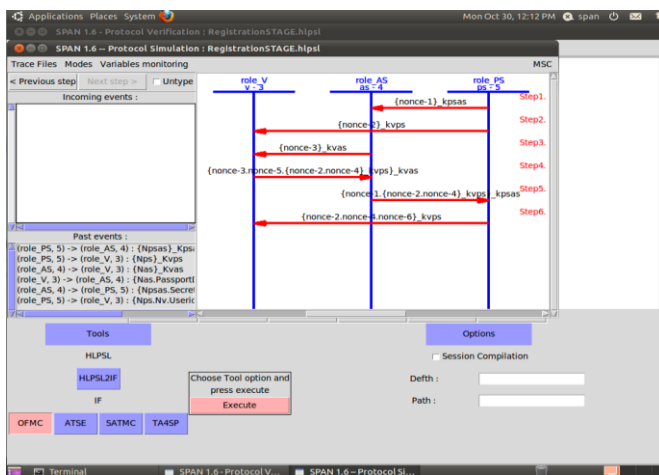


Figure 8 – Registration protocol in "Protocol simulation" mode.

We can see from the simulation results in the field of intercepted data "Intruder knowledge", all transmitted messages are encrypted on keys which intruder doesn't know, and it excludes the possibility in any way to get important information, such as the user's passport data or unique identifier. The record "nonce-N" means some data that is not readable. As a result of the analysis, it was revealed that the registration protocol is safe, ensures the fulfillment of the security objectives (properties) set in the protocol analysis: securing data, authentication of the parties, protection against replay-attacks.

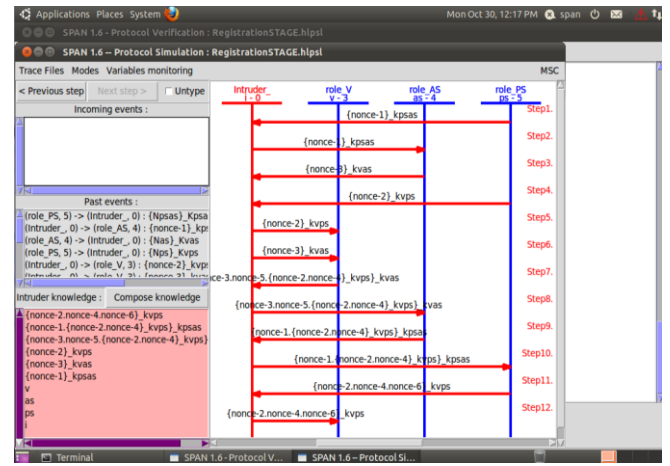


Figure 9 – Registration protocol verification using OFMC mode.

VI. CONCLUSION

The automated security verification tool Avispa was used for security verification of the registration protocol in electronic voting system based on blind intermediaries, in this paper. The protocol was described in the formal languages CAS+ and HLPSSL. The secrecy properties of the transmitted data between the interacting parties were analyzed. It was shown that set security objectives: parties' authentication, verification of data privacy and protection from replay attacks were achieved. The scheme of parties' interaction with the help of tools' graphical functional was considered. An analysis of messages that an intruder can intercept was carried out. Based on the graphical representation it was revealed that all transmitted data is secure, because all messages are encrypted on unknown for intruder keys.

ACKNOWLEDGMENT

The work was supported by the Ministry of Education and Science of the Russian Federation grant № 2.6264.2017/8.9.

REFERENCES

- [1] Overview of e-voting systems, NICK Estonia. - Estonian National Electoral Commission. Tallinn 2005.
- [2] Dossogne J., Lafitte F. Blinded additively homomorphic encryption schemes for self-tallying voting //Journal of Information Security and Applications. - 2015.
- [3] Izabachene M. A Homomorphic LWE Based E-voting Scheme //Post-Quantum Cryptography: 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016
- [4] Hirt M., Sako K. Efficient receipt-free voting based on homomorphic encryption //International Conference on the Theory and Applications of Cryptographic Techniques. - Springer Berlin Heidelberg, 2000. - C. 539-556.
- [5] Rivest L. R. et al. Lecture notes 15: Voting, homomorphic encryption. - 2002.
- [6] Ben Adida, Mixnets in Electronic Voting, Cambridge University (2005)
- [7] Electronic elections: fear of falsification of the results. - Kazakhstan today 2004
- [8] Lipen VY, Voronetsky MA, Lipen DV technology and results of testing electronic voting systems. - United Institute of Informatics Problems NASB 2002.

- [9] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24 (2): 84-90, 1981
- [10] Ali S. T., Murray J. An Overview of End-to-End Verifiable Voting Systems // arXiv preprint arXiv: 1605.08554. - 2016.
- [11] Smart M., Ritter E. True trustworthy elections: remote electronic voting using trusted computing // International Conference on Autonomic and Trusted Computing. - Springer Berlin Heidelberg, 2011. - S.187-202.
- [12] Bruck S., Jefferson D., Rivest R. L. A modular voting architecture ("frog voting") // Toward trustworthy elections. - Springer Berlin Heidelberg, 2010.
- [13] Jonker H., Mauw S., Pang J. Privacy and verifiability in voting systems: Methods, developments and trends // *Computer Science Review*. - 2013.
- [14] Shubhangi S. Shinde, Sonali Shukla, Prof. D. K. Chitre, Secure E-voting Using Homomorphic Technology, *International Journal of Emerging Technology and Advanced Engineering* (2013)
- [15] Neumann S., Volkamer M. Civitas and the real world: problems and solutions from a practical point of view // Availability, Reliability and Security (ARES), 2012 Seventh International Conference on. - IEEE, 2012. - S. 180-185.
- [16] Yi X., Okamoto E. Practical remote end-to-end voting scheme // International Conference on Electronic Government and the Information Systems Perspective. - Springer Berlin Heidelberg, 2011. - S. 386-400.
- [17] The AVISPA team, The High Level Protocol Specification Language, <http://www.avispa-project.org/>, (2006)
- [18] Ronan Saillard, Thomas Genet, CAS+, March 21, 2011
- [19] D. Basin, S. Mödersheim, and L. Viganò. OFMC: A Symbolic Model-Checker for Security Protocols. *International Journal of Information Security*, 2004.
- [20] L.K. Babenko, I.A. Pisarev, O.B. Makarevich. Secure electronic voting using blinded intermediaries. - *Journal "Izvestiya SFedU". Technical sciences*, - Taganrog: Publishing house of ITA SFedU, No. 5, 2017. - p. 6-15.

Towards Formal Verification of Cyber Security Standards

Tomas Kulik
Aarhus University, Denmark
tomaskulik@eng.au.dk

Peter Gorm Larsen
Aarhus University, Denmark
pgl@eng.au.dk

Abstract—Cyber security standards are often used to ensure the security of industrial control systems. Nowadays, these systems are becoming more decentralised, making them more vulnerable to cyber attacks. One of the challenges of implementing cyber security standards for industrial control systems is the inability to verify early that they comply with the standards. In this paper, we propose an approach that uses formal analysis to achieve this. Our approach can be used at an early design stage, where problems are less expensive to correct, to ensure that the system has the desired security properties. We evaluate our approach based on its flexibility to handle and combine different aspects of the cyber security standards.

Index Terms—cyber security, formal analysis, cyber security standards

I. INTRODUCTION

In an industrial setting there is an increasing use of wireless technology because many components becomes Internet of Things (IoT) enabled. Rather than having to investing in a continuation of wired connections the balance between cost and agility many companies moves to such IoT solutions. However, this move towards wireless technologies gives new security challenges that must be taken serious in order to protect both the data and algorithms owned by the companies. In order to ensure this different security standards have emerged and here the IAS/IEC-62443 is a promising candidate that deserves special examination [1].

In order to master the increase of complexity caused by the increased wireless connections the architectures of the distributed systems needs thorough analysis. Here model checking is a promising candidate to provide such an analysis. This has an appropriate balance between the time and cost spent on the analysis and the exhaustiveness favourable. In this paper we demonstrate how this can be achieved defining possible attacks and the corresponding mitigations using a formal approach. The main result is an illustration of how this kind of framework can be deployed to illustrate how a specific architecture and its chosen mitigations can be proven that the different cyber-attacks cannot be realised.

The remainder of this paper is structured as follows: Section II introduces the essential parts of the ISA/IEC-62443 standard and this is followed in Section III defining the architecture of considered system. The main result of this paper is presented in Section IV defining extended formal framework for cyber attacks and possible mitigations for these. Section V explains about how formal analysis can be conducted using

the Alloy Analyzer [2]. This is followed by Section VI, which considers related work for formal analysis of cyber security standards. Finally, Section VII provide concluding remarks also indicating the future directions planned for this work.

II. THE CHOSEN CYBER SECURITY STANDARD

Within this paper we consider security of an industrial control systems based on IoT environment. This is further considered in terms of applying a cyber security standards that are used to ensure industrial automation and control system security, specifically the ISA/IEC-62443 series of standards.

The series is split into 4 distinct groups where each group considers different perspective of cyber security of the industrial automation control system (IACS). Each of the groups contain documents, where each document is understood as a single standard. This leads to name designation of specific standards based on the format: ISA/IEC-62443-X-Y where *X* is the designation of the group and *Y* is the designation of the specific document.

The first group, **ISA/IEC-62443-1, General**, considers the general aspects of the standard and cyber security. Concepts and metrics defined within this group are present throughout the other groups of the standard as shown in Fig. 1. The second group, **ISA/IEC-62443-2, Policies and procedures**, focuses on organizational aspects of cyber security. The main consideration of this group is providing the requirements that the organization has to fulfill in order to manage their cyber security program. The third group, **ISA/IEC-62443-3, System**, addresses the security on a system level. The security requirements for the system is defined here as well as guidance on implementation of these and fulfillment of these requirements. The final group, **ISA/IEC-62443-4, Component**, contains documents defining detailed requirements for cyber security on the component level.

A. The standard under consideration

The standard that we consider for formal verification is ISA/IEC-62443-3-3, System security requirements and security levels. This standard has been selected as it provides requirements that are applicable on system level and are verifiable by technical means. The intended audience for this standard are asset owners, system integrators and service suppliers and the purpose of this standard is to use the defined requirements to evaluate the system under consideration and

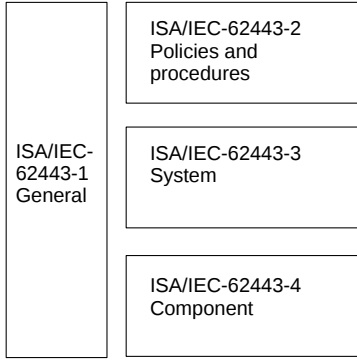


Fig. 1. Overview of ISA/IEC-62443 series structure.

determine if this system is capable of reaching a specific security level (SL). The standard defines 4 SLs:

- **SL 1:** The lowest SL aimed to prevent unauthorized disclosure of information via eavesdropping or casual exposure.
- **SL 2:** Aimed to prevent unauthorized disclosure of information to an entity actively searching for it using simple means with low resources, generic skills and low motivation.
- **SL 3:** Aimed to prevent unauthorized disclosure of information to an entity actively searching for it using sophisticated means with moderate resources, IACS specific skills and moderate motivation.
- **SL 4:** The highest SL aimed to prevent unauthorized disclosure of information to an entity actively searching for it using sophisticated means with extended resources, IACS specific skills and high motivation.

Within the standard the security requirements on the system level are considered as system requirements (SRs) where each SR can define 0 to 3 requirement enhancements (REs). SL of the aspect of the system is measured as a compliance with SRs and REs for this aspect, shown in (Table I).

TABLE I
MAPPING BETWEEN COMPLIANCE WITH SRs, REs AND CORRESPONDING SLs

| SR | RE(s) | SL |
|------|--------------------|------|
| SR 1 | none | SL 1 |
| SR 1 | RE 1 | SL 2 |
| SR 1 | RE 1 + RE 2 | SL 3 |
| SR 1 | RE 1 + RE 2 + RE 3 | SL 4 |

In case that no SR is defined for the given aspect of the system, the standard implicitly defines SL 0 as an SL for this aspect of the system.

III. SYSTEM ARCHITECTURE

The system under consideration extends a generic control systems architecture and capabilities defined in the framework for Threat-driven Cyber Security Verification of IoT

Systems (FCSVIoT) [3]. This architecture consists of subsystems equipped with sensors and actuators shown on Fig. 2. Each subsystem is a microcontroller capable of computation and communication. Communication between the subsystems creates a distributed control system, which provides data to and accepts commands from a central engineering terminal. In this paper we extend the architecture with the notion of **router**, a special type of subsystem that enables data exchange among other subsystems and extends the capabilities of the system by defining user actions on the engineering terminal. We further consider that communication channels must exist between subsystems in order to exchange data.

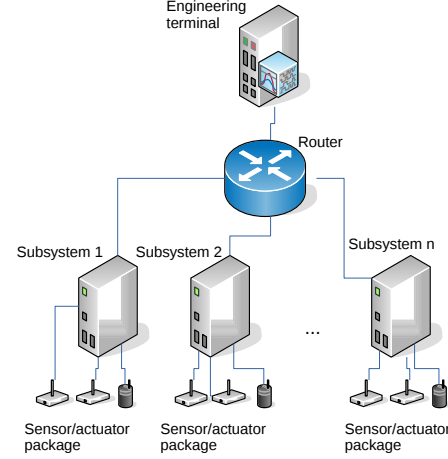


Fig. 2. Architecture of the system under consideration

We let our subsystems be governed by a set of atomic actions forming a basic alphabet for each subsystem S_i as $\mathcal{SA} = \{generate, send, acquire, accept, discard, connect, disconnect, recover, compromise\}$ and each subsystem has a finite set of states S . Actions cause transitions between states of the subsystem such as:

$$s \xrightarrow{action(param)} s' \text{ where } s, s' \in S$$

We further define a predicate on communication channels $secure(c)$ stating that the communication channel is secured. The generate action represents generation of data by the subsystem, send action represents sending the data on a communication channel, acquire represents acquiring data from the communication channel, accept defines accepting the acquired data, discard defines discarding the acquired data. The connect and disconnect action represent a subsystem connecting to and disconnecting from a communication channel. The compromise action moves the subsystem to a compromised mode of operation, $compromised(S_i)$, where we consider that the subsystem has malicious intent. Recover action moves the subsystem from compromised to normal mode of operation, $normal(S_i)$.

We extend the actions in the FCSVIoT by considering the engineering terminal \mathcal{E} as an user interaction part of the system by defining its own alphabet of actions $\mathcal{EA} = \{allow, forbid\}$,

where *allow* represents allowing and *forbid* represents disallowing interaction with an user by the engineering terminal. We also consider that the system holds a set of user accounts allowing users access to the Engineering terminal, $\mathcal{A}c$ where a single account is denoted as a . Each account has exactly one credential cr , hence the system also holds a set of valid credentials Cr . We further define the router \mathcal{R} as with alphabet $\mathcal{RA} = \mathcal{SA} \setminus \{generate\}$ as the router is not equipped with sensors to generate its own data. This leads to creation of system alphabet $\mathcal{A} = \mathcal{SA} \cup \mathcal{EA}$.

IV. ATTACKS AND MITIGATIONS

We define cyber attacks as sequences of events leading to potential harm to the system under attack. Within this paper we consider two cyber attacks, specifically data packet tampering and brute force attack against an user account [4]. These attacks have been purposefully selected as the selected cyber security standard addresses them and specifies requirements for mitigations aimed to increase cost of these attacks. We provide a formal description of the attack sequence and mitigation for both of the attacks under consideration.

1) *Data packet tampering attack*: Packet tampering is the act of a compromised subsystem, specifically a router changing values in a data packet, causing the intended receiving subsystem to receive different values from those sent by the transmitting subsystem. This has then the potential to cause unsafe behavior of the system. In order to describe an instance of this attack, consider two subsystems S_0 and S_1 operating in normal mode, which we show formally as $normal(S_0) \wedge normal(S_1)$ and a router \mathcal{R} used to enable data exchange between the two subsystems. The router operates in a compromised mode $compromised(\mathcal{R})$, meaning that some malicious actor has access to and control over this router.

The subsystems S_0 and S_1 are always connected to the router, meaning that at any time they can exchange data with the router using communication channels c_0 and c_1 . We use the FCSVIoT predicate *always_connected* as $always_connected(S_0, \mathcal{R}_0, c_0) \wedge always_connected(S_1, \mathcal{R}_0, c_1)$, specifying that there is always possibility of communication between S_0 and S_1 via \mathcal{R}_0 .

Now we consider that S_0 is sending a unit of data d to S_1 . The data d is first obtained by \mathcal{R}_0 , which modifies the data d , represented by a new *modify* action added to the alphabet of the router in order to represent software installed by malicious actor, and then sends it further to S_1 . The attack hence combines actions into a pattern by following a specific sequence:

1. $S_0.generate(d)$
2. $S_0.send(c_0, d)$
3. $\mathcal{R}_0.acquire(c_0, d)$
4. $\mathcal{R}_0.modify(d)$
5. $\mathcal{R}_0.send(c_1, d)$
6. $S_1.acquire(c_1, d)$
7. $S_1.accept(d)$

Main act of the attack happens at the $\mathcal{R}_0.modify(d)$ event. Here the data d becomes malicious as $malicious(d)$. In case of non-existent mitigations within the system, the subsystem S_1 simply accepts the data and becomes itself compromised, hence the attack is successful.

In order to mitigate this attack, we consider security requirements from the ISA/IEC-62443-3-3 security standard, covering the communication integrity, namely SR 3.1 stating that *The control system shall provide the capability to protect the integrity of transmitted information*.

The requirement itself does not provide the necessary guidance on what method to use to protect the data, hence we consider the SR 3.1 RE 1 specifying the cryptographic integrity protection as *The control system shall provide the capability to employ cryptographic mechanisms to recognize changes to information during communication*. To mitigate the attack from a general perspective we consider that the data has to contain a cryptographic signature derived from the data content and a secret known to subsystems, but not routers. This introduces a concept of signed data, which we do by extending the alphabet of the subsystem by adding an atomic *sign* event as $sign(d)$. We further define a predicate for signed data stating that the data is considered signed only if signed by a subsystem operating in a normal mode:

$$signed(d) = \exists s : s \xrightarrow{sign(d)}_i s' \wedge s \in S_N$$

We then consider applying the signing event as a mitigation by specifying that the subsystem discards the signed data if it has been modified, with indices added to the state notation, describing the order of state transitions:

$$\begin{aligned} & \forall s_1 : s_1 \xrightarrow{discard(d)}_i s_2 \text{ if} \\ & always_connected(S_i, \mathcal{R}_j, c_k) \\ & \text{and } \exists s_0 : s_0 \xrightarrow{acquire(d, c_k)}_i s_1 \\ & \text{and } \exists s : s \xrightarrow{modify(d)}_j s' \\ & \text{and } signed(d) \end{aligned}$$

Applying this mitigation by adding $S_0.sign(d)$ after the $S_0.generate(d)$ event causes the final event in the chain to be $S_1.discard(d)$ since $\mathcal{R}_0.modify(d)$ is present. This means that the subsystem S_1 does not enter compromised mode and the cyber attack is unsuccessful.

2) *Brute force attack against an user account*: Brute force attack against an user account uses computational power to try to guess user sign in credentials by randomly generating passwords and user names and providing them to the system for verification. The attack can be streamlined if the user name and length of the password is known, decreasing the "guess space", which in turn leads to less time required to guess the correct credentials. If the user account can be breached this gives the malicious actor control over the system in terms that the breached account allows, potentially allowing the malicious actor submission of malicious commands to

the system. To formally describe the attack we consider a single engineering terminal \mathcal{E}_0 operating in a normal mode, $normal(\mathcal{E}_0)$. We further define a *check* function on an engineering terminal, responsible for raising the *allow* or *forbid* event:

$$check(cr) = \begin{cases} allow, & \text{if } cr \in Cr \\ forbid, & \text{otherwise} \end{cases}$$

We formulate the attack as a recursive *crack*(*cr*) function that generates new *cr* for every attempt used to find a *cr* such that $cr \in Cr$:

$$crack(check(cr)) = \begin{cases} true, & \text{if } allow \\ crack(check(new(cr))), & \text{if } forbid \end{cases}$$

Once the function returns true the malicious actor has obtained access to the engineering terminal, causing the engineering terminal to operate in a compromised mode of operation, as $compromised(\mathcal{E}_0)$ and the attack is considered successful.

In order to mitigate the attack we consider the requirement SR 1.11 defined in ISA/IEC-62443-3-3, stating, *The control system shall provide the capability to enforce a limit of a configurable number of consecutive invalid access attempts by any user (human, software process or device) during a configurable time period. The control system shall provide the capability to deny access for a specified period of time or until unlocked by an administrator when this limit has been exceeded.* To enforce this we define a locked predicate acting on specific account mapped via its valid credential where mapping between account *ac* and credential *cr* is one to one and hence for simplicity we omit *cr* and consider *ac* as belonging to a specific *cr* as:

$$locked(ac) = \neg \exists s : s \xrightarrow{allow()} s' : ac \in Ac$$

We then need to consider the amount of allowed invalid access attempts. In order to abstract away from details of password complexity, we present an assumption stating that the successful brute force attack against an system that allows reasonable small amount (in general we would consider this less than 10 for practical reasons) invalid access attempts is so unlikely that we consider it impossible. Using this assumption as a mitigation we can guarantee that the user account cannot be breached by brute force attack. We also abstract away from notion of time intervals as we consider that the brute force attack is happening rapidly and would always exceed the amount of tries within a specific time interval. We formally show this mitigation by first defining a global variable for *ac* holding the current attempt as *attempt*(*ac*) for its credential *cr*:

$$attempt(ac) = \begin{cases} attempt(ac) + 1, & \text{if } check(cr) = forbid \\ 0, & \text{otherwise} \end{cases}$$

We then use the variable in adding an attempt limit on using a credential to sign in to an user account such that the

account becomes locked if the maximum amount of attempts is reached:

$$limited(cr, max_att) = locked(ac) \text{ if } attempt \geq max_att$$

By applying the *limited* predicate to the credentials we cause the account to become *locked* as a result of the *crack* function. Since a locked account cannot be used to gain access to the engineering terminal, the cyber attack fails and the engineering terminal continues in the normal mode of operation, $normal(\mathcal{E}_0)$. It is important to note that in general the *max_attempt* has to be set in such a way that does not hinder usability of the system, while providing assurance of sufficient security. This mitigation strategy has therefore a limitation in case the *max_attempt* is set unreasonably large.

V. FORMAL ANALYSIS

In this section we shortly present the extensions made to the FCSViOT and show how the mitigations for data packet tampering and brute force against user account attacks have been verified when considered within the architecture defined in Section III. This is achieved by expressing the aforementioned attacks and mitigations using FCSViOT with extensions introduced in this paper and verifying these scenarios using the Alloy Analyzer.

1) *Short introduction to Alloy Analyzer:* Alloy is a formal specification language, based on first order logic, used for expressing structural constraints in software systems. Alloy allows for modeling at different levels of abstraction, where at the highest level it provides object oriented interpretation, at second level it uses the set theory and at the lowest level atoms and relations are used. Within our model we are using the set theory, atoms and relations to model the types using the **sig** keyword. Subtyping is supported in Alloy by usage of **extends** keyword. We model relations between objects by specifying mappings between sets, for example **has:set** EngTerminal **one**->**some** Account, where **has** is the relation stating that the **one**, meaning exactly one engineering terminal has **some**, meaning at least one account associated with it. The scope of the model is specified after the **run** block, by quantifying how many atoms do we want to include in the model by using the **exactly** keyword. Properties of the Alloy model can be verified by usage of the Alloy Analyzer software tool [5], which checks properties of the model by generating counterexamples.

2) *Overview of extensions to FCSViOT:* Among the first extensions is addition of new data types Router corresponding to the router, EngTerminal corresponding to the engineering terminal, Account corresponding to user account and Credential corresponding to credential for specific account as specified in Section III. Using Alloy Analyzer, we define these datatypes using set definitions, represented by the **sig** keyword. We further extend the State definition with number of new relations. We further define the router as an extension of Device type and also adapt Subsystem to be extension of Device, as shown in Listing 1.

Listing 1. Extensions and changes to the modeling framework

```

open util/ordering[State]
sig Data {}
sig Device {}
sig Subsystem extends Device{}
sig Router extends Device{}
sig Channel {}
sig EngTerminal {}
sig Credential{}
sig Account{}
sig State {
  ...
  compromised: set Device,
  can_authorise: set Subsystem,
  malicious: set Data,
  signed: set Data,
  accepted: set Device -> set Data,
  secure: set Channel,
  attempts_exceeded: set Account,
  limited: set Account,
  cracked: set Account,
  large: set Credential,
  locked: set Account,
  has: set EngTerminal one->some Account,
  hasCred: set Account one->one Credential
  ...
}{ /* Facts belonging to State */ ... }

```

The Device type is used as a base type since Router and Subsystem share most of the actions. The only difference is that we consider that the router is not capable of generating data. The relations within State now also use Device in order to model relation that cover both Router and Subsystem. For example, the `compromised` relation shown in Listing 1 shows that a state can contain any number of compromised devices. Another relation recorded in each state is for example `accepted` which maps devices to data.

The different types discussed above are governed by several facts, which are understood as constraints on the model. One of these is the consideration that the data is either signed or not and this does not change as the system progresses in its state transitions. This is shown in Listing 2. In this constraint `s'` is the state following the `s` state, hence the constraint guarantees that the data remains signed in all states.

Listing 2. Global constraint governing signed data

```

fact{
  ...
  all s:State, s':s.next |
    s.signed = s'.signed
}

```

3) *Verification of data tampering mitigation strategy:* Here we demonstrate the mitigation strategy applied to a scenario discussed in IV-1. The simplest model to demonstrate data tampering mitigation strategy in fact only requires one subsystem and a router as it is the router that is responsible for

the attack. This is shown in Listing 3. The listing shows the constraint for mitigation and the setup of the model. The complete extended FCSVIoT can be found via [6].

Listing 3. Verifying the data tampering mitigation strategy using Alloy

```

run {
  ...
  // mitigation signed data
  all s:State | all d:Data | d in s.signed
  //test the condition
  some malicious
  ...
} for
exactly 5 State
, exactly 1 Subsystem
, exactly 1 Data
, exactly 1 Channel
, exactly 1 Router
, exactly 2 Device
, exactly 0 EngTerminal
...

```

The `run` commands checks that the data is signed in five states, required to execute the whole scenario. The result of this execution is: No instance found. This means that the Alloy Analyzer could not find a counter-example within the requested scope and the mitigation strategy is proven to work.

4) *Verification of brute force attack:* We will show how the mitigation strategy for brute force attack can be modeled by considering a scenario described in Section IV-2. The `run` command for the model we use consists of one engineering terminal with one user account, with its associated credentials and omits subsystems as shown in Listing 4. The mitigation used in this scenario states that all accounts within all states of the system are always considered limited (i.e. they consider limit on the number of unsuccessful login attempts).

Listing 4. Verifying the brute force attack mitigation strategy using Alloy

```

run {
  ...
  // mitigation account has limited tries
  all s:State | all a:Account |
    a in s.limited
  //test the condition
  some cracked
  //start with no cracked account
  no first.cracked
  ...
} for
exactly 3 State
, exactly 1 EngTerminal
, exactly 1 Account
, exactly 1 Credential
, exactly 0 Subsystem
...

```

This scenario considers three states, creating the smallest scope necessary for its execution. Once the `run` command is executed the Alloy Analyzer returns `No instance found`, confirming that the mitigation strategy prevents the user account from being cracked.

VI. RELATED WORK

As cyber security is becoming very important topic in the industry, mainly in advent of digitalization and trends such as industry 4.0 [7], research is being carried out within the area of using formal methods in order to provide proofs that systems meet cyber security requirements [8]–[10]. The benefits of using model based verification are its applicability at an early stage of system development in order to help avoid exposure to attacks as well as provide mitigations for attacks that are not easily avoidable [11], [12]. This approach consists of formal description of the behavior of a system and formal description of cyber attacks and mitigations. The complete model is then formally analysed in order to verify that the mitigation strategies prevent the cyber attacks from causing potentially harmful behavior of the system. Sometimes specific cyber security standards are considered as criteria for these mitigations strategies [13].

In order to provide assurance that an industrial control system meets criteria specified in a cyber security standard, authors of [14] have investigated the ISA-99.01-01 standard by considering the requirements and metrics specified within the standard. While the authors have described part of the standard formally, their goal was not to conduct formal analysis to ensure the satisfiability of the security requirements by a given architecture but rather to provide recommendations to the operators of industrial control systems to not blindly trust standards but verify their security impact on the system.

The authors of [15] have proposed a formalization and verification technique for ISO/IEC-15408 standard known as Common Criteria using Z notation. In their technique they consider the natural language definitions within the standard and create formal templates based on these. The authors suggest usage of the templates against the formalized specification of the target system, which is left to the party verifying the system against the instantiated templates. The authors provide an example of this verification using the Z/EVES theorem prover. Our approach differs by providing formal building blocks for the system from the start, hence formalization of the system can be done by selecting from these building blocks.

VII. CONCLUSIONS AND FUTURE WORK

So far this research has demonstrated that the chosen approach is quite extensible where this paper has demonstrated how the models made in Alloy can be extended in a conservative manner with additional threats. It is expected that we in the future in this context is furthering the formal definitions to encompass more of aspects of the security standard and to verify these against larger variety of cyber attacks. We further consider switching to TLA+ [16] in order to show the applicability of our framework using different formalism.

VIII. ACKNOWLEDGMENTS

This work is partially supported by the Manufacturing Academy of Denmark (MADE) Digital project. For more information see <http://www.made.dk/>.

REFERENCES

- [1] International Society of Automation, “The 62443 Series of Standards,” <http://isa99.isa.org/Public/Information/The-62443-Series-Overview.pdf>, accessed on 13/3/18.
- [2] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Heyward Street, Cambridge, MA02142, USA: MIT Press, April 2006, ISBN-10: 0-262-10114-9.
- [3] Tomas Kulik and Peter W. V. Tran-Jørgensen and Jalil Boudjadar and Carl Schultz, “A framework for threat-driven cyber security verification of iot systems,” april 2018, First International Workshop on Verification and Validation of Internet of Things, Västerås, Sweden, In print.
- [4] C. Bekara, “Security issues and challenges for the iot-based smart grid,” *Procedia Computer Science*, vol. 34, pp. 532 – 537, 2014, the 9th International Conference on Future Networks and Communications (FNC’14)/The 11th International Conference on Mobile Systems and Pervasive Computing (MobiSPC’14)/Affiliated Workshops. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050914009193>
- [5] “The Alloy Analyzer Modelling website,” <http://alloy.mit.edu/alloy/>, 2018.
- [6] Tomas Kulik and Peter Gorm Larsen, “Extensions to formal security modeling framework,” <https://github.com/kuliktomas/FCSVioT/commit/189c7962f7f0870fa5315c31a71a6b35e896e47d>, 2018.
- [7] N. Jazdi, “Cyber physical systems in the context of industry 4.0,” in *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*, May 2014, pp. 1–4.
- [8] M. Ge and D. S. Kim, “A framework for modeling and assessing security of the internet of things,” in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 776–781.
- [9] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean, “Applying Formal Methods to a Certifiably Secure Software System,” *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 82–98, jan.-feb. 2008.
- [10] A. N. Haidar and A. E. Abdallah, “Formal modelling of pki based authentication,” *Electronic Notes in Theoretical Computer Science*, vol. 235, pp. 55 – 70, 2009, proceedings of the 4th International Workshop on Automated Specification and Verification of Web Systems (WWV 2008). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S157106610900084X>
- [11] D. C. Wardell, R. F. Mills, G. L. Peterson, and M. E. Oxley, “A method for revealing and addressing security vulnerabilities in cyber-physical systems by modeling malicious agent interactions with formal verification,” *Procedia Computer Science*, vol. 95, no. Supplement C, pp. 24 – 31, 2016, complex Adaptive Systems Los Angeles, CA November 2-4, 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050916324619>
- [12] F. A. Teixeira, F. M. Pereira, H.-C. Wong, J. M. Nogueira, and L. B. Oliveira, “Siot: Securing internet of things through distributed systems analysis,” *Future Generation Computer Systems*, 2017. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167739X17304235>
- [13] J. Woodcock, S. Stepney, D. Cooper, J. A. Clark, and J. Jacob, “The Certification of the Mondex Electronic Purse to ITSEC Level E6,” *Formal Aspects of Computing*, vol. 20, no. 1, pp. 5–19, 2008.
- [14] D. K. Holstein and K. Stouffer, “Trust but verify critical infrastructure cyber security solutions,” in *2010 43rd Hawaii International Conference on System Sciences*, Jan 2010, pp. 1–8.
- [15] S. Morimoto, S. Shigematsu, Y. Goto, and J. Cheng, “Formal verification of security specifications with common criteria,” in *Proceedings of the 2007 ACM Symposium on Applied Computing*, ser. SAC ’07. New York, NY, USA: ACM, 2007, pp. 1506–1512. [Online]. Available: <http://doi.acm.org/10.1145/1244002.1244325>
- [16] L. Lamport, *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

On the model checking of finite state transducers over semigroups

Anton Gnatenko

Lomonosov Moscow State University

Moscow, Russia

gnatenko.cmc@gmail.com

Vladimir Zakharov

National Research University Higher School of Economics

Moscow, Russia

zakh@cs.msu.su

Abstract—Sequential reactive systems represent programs that interact with the environment by receiving signals or requests and react to these requests by performing operations with data. Such systems simulate various software like computer drivers, real-time systems, control procedures, online protocols, etc. In this paper we study the verification problem for the programs of this kind. We use finite state transducers over semigroups as formal models of reactive systems. We introduce a new specification language LP-CTL* to describe the behavior of transducers. This language is based on the well-known temporal logic CTL* and has two distinguished features: 1) each temporal operator is parametrized with a regular expression over input alphabet of the transducer, and 2) each atomic proposition is specified by a regular expression over the output alphabet of the transducer. We develop a tabular algorithm for model checking of finite state transducers over semigroups against LP-CTL* formulae, prove its correctness, and estimate its complexity. We also consider particular fragments of LP-CTL* language, where temporal operators are parametrized with regular expressions over special alphabets, and show that these fragments may be used to specify usual Kripke structures, while they are more expressive than usual CTL*.

I. INTRODUCTION

Finite state machines are widely used in the field of computer science and formal methods for various purposes. While finite automata represent regular sets, *transducers* stand for regular (or, rational) *relations* and, therefore, can serve as models of programs and algorithms that operate with input and output data. For example, transducers are used as formal models in software engineering to represent numerous algorithms, protocols and drivers that manipulate with strings, dataflows, etc [1], [15], [25].

By extending the concept of ordinary transducers we build a new formal model for sequential reactive systems. These systems are software programs or hardware devices that receive requests (control signals, commands) from the environment and perform in response some manipulations (actions, transformations) with data, interactions with the environment, mechanical movements, etc. While the flow of requests can be represented by finite or infinite words in some fixed alphabet, the sequence of actions of the system needs a more sophisticated interpretation. The key point here is that different sequences of actions may bring a computing system to the same result. To capture this effect the collection of actions performed by a reactive system can be viewed as a

generating set of some algebraic structure (e.g. semigroup, group, ring, etc.) and particular algebraic properties of basic actions should be taken into account when choosing adequate formal models for this class of information processing systems. Let us illustrate this consideration by several examples.

- A network switch with several input and output ports. A switch is a device which receives data packets on its input port, modifies their heads and commutes them to one of the output ports. Once received a special control signal, this switch changes its packet forwarding table and, thus, its behaviour. Since packets from different flows may be processed in any order, the switch can be modeled by a transducer which operates over a free partially commutative semigroup, or a trace monoid. Trace monoids are commonly used as an algebraic foundation of concurrent computations and process calculi (see, e.g., [9]).
- A real-time device that control the operation of some industrial equipment (say, a boiling system). Such device receives data from temperature and pressure sensors and switches some processes on and off according to its instructions and the current state of the system. It seems reasonable that for some actions the order of their implementation is not important (routine actions), while others must follow in a strictly specified order (e. g. an execution of some complex operation). Moreover, there are also actions which bring system to certain predefined operation mode (set-up actions). These actions are implemented in the emergency situations. A partially commutative semigroup with right-zero elements $\mathbf{0}$ which satisfy the equalities $x\mathbf{0} = \mathbf{0}$ for every element x provides an adequate interpretation for such operations.
- A system supervisor that maintains a log-file. For each entry its date and time is recorded in the file and there is no way to delete entries from the log — only to append it. Thus, for any two basic actions (record operations to the log-file) it is crucial in which order they are performed and such a supervisor can be modeled by a transducer over a free semigroup. Verification techniques for such reactive systems are considered in [17]; this is the main topic of this paper as well.
- A radio-controlled robot that moves on the earth or moon surface. It can make one step moves in any of direction.

When it receives a control signal in a state q' it must choose and carry out a sequence of steps and enter to the next state q'' . At some distinguished state q_f the robot reports its current location. Movements of the robot may be regarded as basic actions, and the most simple model of computation which is suitable for analyzing a behaviour of this robot is a nondeterministic finite state transducer operating on a free Abelian group of rank 2.

To construct a reliable system or network it is crucial for its components to have a correct behaviour. For example, a network switch must process received data packets within a specified number of execution steps and the boiling system should never be overheated, that is, will never remain for a long time in a particular condition without appropriate responses from the control device. By using finite state transducers as formal models of reactive systems one can develop verification algorithms for these models to solve such problems as equivalence checking, deductive verification or model checking.

The study of the equivalence checking problem for classical transducers began in the early 60s. It was established that the equivalence checking problem for non-deterministic transducers is undecidable [13] even over 1-letter input alphabet [16]. However, the undecidability displays itself only in the case of unbounded transductions when an input word may have arbitrary many images. The equivalence checking problem was shown to be decidable for deterministic [4], functional (singlevalued) [5], [19], and k -valued transducers [6], [26]. In a series of papers [20], [21], [22] techniques for checking bounded valuedness, k -valuedness and equivalence of finite state transducers over words were developed. Recently in [29] equivalence checking problem was shown to be decidable for finite state transducers that operate over finitely generated semigroups embeddable in decidable groups.

There are also papers where equivalence checking problem for transducers is studied in the framework of program verification. The authors of [23] proposed models of communication protocols as finite state transducers operating on bit strings. They set up the verification problem as equivalence checking between the protocol transducer and the specification transducer. The authors of [25] extended finite state transducers with symbolic alphabets which are represented as parametric theories. They showed that a number of classical problems for extended transducers, including equivalence checking problem, are decidable modulo underlying theories. In [1] a model of streaming transducers was proposed for programs that access and modify sequences of data items in a single pass. It was shown that a number of verification problems such as equivalence checking, assertion checking, and checking correctness with respect to pre/post conditions, are decidable for this program model.

Meanwhile, very few papers on the model checking problem for transducers are known. Transducers can be conveniently used as auxiliary means in regular model checking of parameterized distributed systems where configurations are represented as words over a finite alphabet. In such models

a transition relation on these configurations may be regarded as a rational relation and, thus, it may be specified by finite state transducers (see [7], [28]). In these papers finite state transducers just play the role of verification instrument, but not an object of verification. But, as far as we know, a deeper investigation of the model checking problem for the reactive systems represented by transducers has not yet been carried out. We think that this is due the following main reason. A transducer is a model of computation which, given a word in an input alphabet, computes a word in an output alphabet. The letters of input and output alphabets can be regarded as valuations (tuples of truth values) of some set of basic predicates. Therefore, a transducer can be viewed as some special representation of a labeled transition system (Kripke structure) (see [2]). From this viewpoint model checking problem for finite state transducers conforms well to standard model checking scheme for finite structures, and, hence, it is not worthy of any particular treatment.

But our viewpoint is quite different. Transducer is a more complex model of computation than a finite state automaton (transition systems). Its behaviour is characterized by the correspondence between input and output words. A typical property of such behaviour to be checked is whether for every (some) input word from a given pattern a transducer outputs a word from another given pattern. Therefore, when formally expressing the requirements of this kind one needs not only temporal operators to specify an order in which events occur but also some means to refer to such patterns. Conventional Temporal Logics like *LTL* or *CTL* are not sufficient in this case; they should be modified in such a way as to acquire an ability to express correspondences between the sets (languages) of input words and the sets (languages) of output words. This could be achieved by supplying temporal operators with patterns as parameters. Every such pattern is a formal description of a language L over an input alphabet \mathcal{C} ; automata, formal grammars, regular expressions, language equations are suitable for this purpose. The basic properties of output words can be also represented by languages over an output alphabet. Then, for instance, an expression $\mathbf{G}_L P$ can be understood as the requirement that for every input word w from the language L the output word h of a transducer belongs to the language P .

The advantages of this approach are twofold. On the one hand, such extensions of Temporal Logics make it possible to express explicitly relationships between input and output words and specify thus desirable behaviours of transducers. On the other hand, it can be hoped that such extensions could rather easily assimilate some well-known model checking techniques (see [8], [3]) developed for conventional Temporal Logics. The first attempt to implement this approach was made in [17]. The authors of this paper introduced an \mathcal{LP} -*LTL* specification language based on *LTL* temporal logic and developed a checking procedure for transducers over free monoids against specifications from \mathcal{LP} -*LTL*. It was shown that this procedure has double exponential time complexity.

In this paper we continue this line of research and "raise"

the specification language introduced in [17] to the level of $\mathcal{LP}\text{-}CTL^*$. We will focus only on one task related to the use of new logic for the verification of reactive systems, namely, the development of a general model checking algorithm for finite state transducers against specifications in $\mathcal{LP}\text{-}CTL^*$. Such issues as expressive power of $\mathcal{LP}\text{-}CTL^*$, complexity of model checking and satisfiability checking problems, the influence of types of languages used as parameters and basic predicates in $\mathcal{LP}\text{-}CTL^*$ on decidability and complexity of model checking problem remain topic of our further research and will be covered in our subsequent works. We also leave beyond this work a number of applied questions, which are worthy of consideration in a separate paper. For example, it is important to understand to what extent the already developed model checking tools can be adapted to the new temporal logic. And, of course, in the future we will have a well-chosen series of examples that illustrate the new possibilities of using $\mathcal{LP}\text{-}CTL^*$ to describe the behavior of reactive systems.

The paper is organized as follows. In Section II we define the concept of finite state transducer over semigroup as a formal model of sequential reactive systems (see [29]) and in Section III we describe the syntax and the semantics of $\mathcal{LP}\text{-}CTL^*$ as a formal language for specifying behaviour of sequential reactive systems. In Section III we also set up formally model checking problem for finite state transducers against $\mathcal{LP}\text{-}CTL^*$ formulae. In Section IV we present an $\mathcal{LP}\text{-}CTL^*$ model checking algorithm for the case when parameters of temporal operators and basic predicates are regular languages represented by finite state automata. The model checking algorithm we designed has time complexity which is linear of the size of a transducer but exponential of the size of $\mathcal{LP}\text{-}CTL^*$ formula. This complexity estimate is in contrast to the case of conventional CTL model checking: its time complexity is linear of both the size of a model and the size of a CTL formula. To explain this effect in Section V we show how $\mathcal{LP}\text{-}CTL^*$ formulae can be also checked on the conventional Kripke structures. Finally, we compare $\mathcal{LP}\text{-}CTL^*$ with some other known extensions of Temporal Logics and discuss some topics for further research.

II. FINITE STATE TRANSDUCERS AS MODELS OF REACTIVE SYSTEMS

In this section we introduce a Finite State Transducer as a formal model of a reactive computing system which receives control signals from the environment and reacts to these signals by performing operations with data.

Let \mathcal{C} be a finite set of *signals*. Finite words over \mathcal{C} are called *signal flows*; the set of all signal flows is denoted by \mathcal{C}^* . Given a pair of signal flows u and v we write uv for their concatenation, and denote by ε the empty flow.

Let $\mathcal{A} = \{a_1, \dots, a_n\}$ be a finite set of elements called *basic actions*; these actions stand for the elementary operations performed by a reactive system. Finite words over \mathcal{A} are called *compound actions*; they denote sequential compositions of basic actions. Since different sequences of basic actions could produce the same result, one may interpret compound

actions over a semigroup (S, e, \circ) generated by a set of basic actions \mathcal{A} . The elements of S are called *data states*. Every compound action $h = a_{i_1}a_{i_2}\dots a_{i_k}$ is evaluated by the data state $[h] = a_{i_1} \circ a_{i_2} \circ \dots \circ a_{i_k}$. For example, if a reactive system just keeps a track of input requests by adding certain records to a log-file then a free semigroup will be suitable for interpretation of these operations. But when a robot moves on a 2-dimensional surface then the actions (movements) performed by this robot may be regarded as generating elements of Abelian group G of rank 2, and the positions on the surface occupied by this robot can be specified by the elements from G . In this paper we restrict ourselves to the consideration of free semigroups when $[h] = h$ holds for every compound action h , and \circ is the word concatenation operation.

Let \mathcal{C} be a set of signals and \mathcal{A} be a set of basic actions that are interpreted over a semigroup (S, e, \circ) . Then a *Finite State Transducer* (in what follows, FST) is a quintuple $\Pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$, where

- Q is a finite set of *control states*;
- $q_{init} \in Q$ is an *initial control state*;
- $T \subseteq Q \times \mathcal{C} \times Q \times \mathcal{A}^*$ is a finite transition relation.

Each tuple (q', c, q'', h) in T is called a *transition*: when a transducer is in a control state q' and receives a signal c , it changes its state to q'' and performs a compound action h . We denote such transition by $q' \xrightarrow{c, h} q''$. A *run* of a FST Π is any finite sequence of transitions

$$q_1 \xrightarrow{c_1, h_1} q_2 \xrightarrow{c_2, h_2} \dots \xrightarrow{c_n, h_n} q_{n+1};$$

this run transduces a signal flow $w = c_1c_2\dots c_n$ into a data state $[h_1h_2\dots h_n]$.

The behaviour of a FST $\Pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ over a semigroup (data space) (S, e, \circ) is presented formally by a *transition system* $TS(\Pi, S) = (D, \mathcal{C}, d_{init}, \mathcal{T})$, where

- $D = Q \times S$ is (in general case, infinite) set of *states of computation*,
- $d_{init} = (q_{init}, e)$ is the *initial state*, and
- $\mathcal{T} \subseteq D \times \mathcal{C} \times D$ is a *transition relation* such that for every states of computation $d' = (q', s')$, $d'' = (q'', s'')$ and every signal $c \in \mathcal{C}$ the relationship

$$(d', c, d'') \in \mathcal{T} \iff \exists h \in \mathcal{A}^*: (q', c, q'', h) \in T \text{ and } s'' = s' \circ [h]$$

holds.

As usual, a transition $(d', c, d'') \in \mathcal{T}$ is denoted by $d' \xrightarrow{c} d''$.

A *trajectory* in a transition system $TS(\Pi, S)$ is a pair $tr = (d_0, \alpha)$, where $d_0 \in D$ and

$$\alpha = (c_1, d_1), (c_2, d_2), \dots, (c_i, d_i), \dots$$

is a sequence of pairs (c_i, d_i) such that $d_{i-1} \xrightarrow{c_i} d_i$ holds for every i , $i \geq 1$. A trajectory represents a possible scenario of a behaviour of a sequential reactive system: when receiving a signal flow $c_1, c_2, \dots, c_i, \dots$ the reactive system performs a sequence of basic actions h and follows sequentially via the states of computation $d_1, d_2, \dots, d_i, \dots$. By $tr|_i$ we mean the trajectory $(d_i, \alpha|_i)$, where $\alpha|_i = (c_{i+1}, d_{i+1}), (c_{i+2}, d_{i+2}), \dots$ is a suffix of α .

III. \mathcal{LP} -CTL* SPECIFICATION LANGUAGE

When designing sequential reactive systems one should be provided with a suitable formalism to specify the requirements for their desirable behaviour. For example, one may expect that

- a mobile robot, receiving an equal number of control signals "go_left" and "go_right", will always return to its original position,
- a network switch will never commute data packets from different packet flows into the same output buffer,
- it is not possible for the interrupt service routine to complete the processing of one interrupt before it receives a request to handle another.

These and many other requirements which refer to the correspondences between control flows and compound actions in the course of FST runs can be specified by means of Temporal Logics. When choosing a suitable temporal logic as a formal specification language of FST behaviours one should take into account two principal features of our model of sequential reactive systems:

- 1) since a FST operates over a data space which is semi-group, the basic predicates must be interpreted over semigroups as well, and
- 2) since a behaviour of a FST depends not on the time flow itself but on a signal flow which it receives as an input, temporal operators must be parameterized by certain descriptions of admissible signal flows.

To adapt traditional temporal logic formalism to these specific features of FST behaviours the authors of [17] introduced a new variant of Linear Temporal Logic (LTL). We assume that in general case one may be interested in checking the correctness of FST's responses to arbitrary set of signal flows. Every sets of control flows may be regarded as a language over the alphabet \mathcal{C} of signals. Therefore, it is reasonable to supply temporal operators ("globally" G , "eventually" F , etc.) with certain descriptions of such languages as parameters. In more specific cases we may confine ourselves with considering only a certain family of languages (finite, regular, context-free, etc.) \mathcal{L} used as parameters of temporal operators. These languages will be called *environment behaviour patterns*.

A reactive system performs finite sequences of basic actions in response to control signals from the environment and thus follows in the course of its run via a sequence of data states which are elements of a semigroup (S, e, \circ) . Therefore, basic predicates used in LTL formulae may be viewed as some sets of data states $S', S' \subseteq S$. These sets can be also specified in language-theoretic fashion. Any language P over the alphabet of basic actions \mathcal{A} corresponds to a predicate (set of data states) $S_P = \{[h] \mid h \in P\}$. As in the case of environment behaviour patterns we may distinguish a certain class \mathcal{P} of languages and use them as specifications of basic predicates. When these languages are used as parameters in temporal formulae then it will be assumed that they are defined constructively by means of automata, grammars, Turing machines, etc.

Thus, we arrive at the concept of \mathcal{LP} -variants of Temporal Logics. In [17] the syntax and semantics of \mathcal{LP} -LTL was studied in some details in the case when both environment behaviour patterns and basic predicates are regular languages presented by finite automata. In this paper we make one step further and extend the concept of \mathcal{LP} -variants of Temporal Logics to CTL^* . Select an arbitrary family of environment behaviour patterns \mathcal{L} and a family of basic predicates \mathcal{P} . The set of \mathcal{LP} -CTL* formulae consists of *state formulae* and *trajectory formulae* which are defined as follows:

- 1) each basic predicate $P \in \mathcal{P}$ is a state formula;
- 2) if φ_1, φ_2 are state formulae then $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$ are state formulae;
- 3) if ψ is a trajectory formula then $\mathbf{A}\psi$ and $\mathbf{E}\psi$ are state formulae;
- 4) if φ is a state formula then φ is a trajectory formula;
- 5) if ψ_1, ψ_2 are trajectory formulae then $\neg\psi_1$ and $\psi_1 \wedge \psi_2$ are trajectory formulae;
- 6) if $\varphi, \varphi_1, \varphi_2$ is a state formula, $c \in \mathcal{C}$, and $L \in \mathcal{L}$ then $\mathbf{X}_c\varphi$, $\mathbf{Y}_c\varphi$, $\mathbf{F}_L\varphi$, $\mathbf{G}_L\varphi$, and $\varphi_1 \mathbf{U}_L\varphi_2$ are trajectory formulae.

The specification language \mathcal{LP} -CTL* is the set of all state formulae constructed as defined above.

Now we introduce the semantics of \mathcal{LP} -CTL* formulae. These formulae are interpreted over transition systems. Let $M = TS(\Pi, S)$ be a transition system, d be a state of computation in this system, and tr be a trajectory in M . Then for every state formula φ we write $M, d \models \varphi$ to denote the fact that the assertion φ is true in the state d of M , and for every trajectory formula ψ we write $M, tr \models \psi$ to denote the fact that the assertion ψ holds for the trajectory tr in M .

In the definition below it is assumed that M is a transition system, $d = (q, s)$ is a state of computation in M , and $tr = (d_0, \alpha)$ is a trajectory in M such that $\alpha = (c_1, d_1), (c_2, d_2), \dots, (c_i, d_i), \dots$. We define the satisfiability relation \models by induction on the height of formulae:

- 1) $M, d \models P \iff s \in P$;
- 2) $M, d \models \neg\varphi \iff$ it is not true that $M, d \models \varphi$;
- 3) $M, d \models \varphi_1 \wedge \varphi_2 \iff M, d \models \varphi_1$ and $M, d \models \varphi_2$;
- 4) $M, d \models \mathbf{E}\varphi \iff$ there exists a trajectory $tr' = (d, \alpha')$ in M such that $M, tr' \models \varphi$;
- 5) $M, d \models \mathbf{A}\varphi \iff$ for any trajectory $tr' = (d, \alpha')$ in M it is true that $M, tr' \models \varphi$;
- 6) if φ is a state formula then $M, tr \models \varphi \iff M, d_0 \models \varphi$;
- 7) $M, tr \models \neg\psi \iff$ it is not true that $M, tr \models \psi$;
- 8) $M, tr \models \psi_1 \wedge \psi_2 \iff M, tr \models \psi_1$ and $M, tr \models \psi_2$;
- 9) $M, tr \models \mathbf{X}_c\varphi \iff c = c_1$ and $M, d_1 \models \varphi$;
- 10) $M, tr \models \mathbf{Y}_c\varphi \iff$ either $c \neq c_1$, or $M, d_1 \models \varphi$;
- 11) $M, tr \models \mathbf{F}_L\varphi \iff \exists i \geq 0: c_1c_2 \dots c_i \in L$ and $M, tr|_i \models \varphi$;
- 12) $M, tr \models \mathbf{G}_L\varphi \iff \forall i \geq 0: \text{ if } c_1c_2 \dots c_i \in L \text{ then } M, tr|_i \models \varphi$;
- 13) $M, tr \models \varphi \mathbf{U}_L\psi \iff \exists i \geq 0: c_1c_2 \dots c_i \in L$ such that $M, tr|_i \models \psi$ and $\forall j, 0 \leq j < i, \text{ if } c_1c_2 \dots c_j \in L$ then $M, tr|_j \models \varphi$.

Observe, that operators \mathbf{X}_c and \mathbf{Y}_c , as well as \mathbf{F}_L and \mathbf{G}_L , are dual to each other:

Proposition 1. *For any $\mathcal{LP}\text{-CTL}^*$ formula φ , any $c \in \mathcal{C}$ and any $L \in \mathcal{L}$, and for an arbitrary trajectory tr in M*

- 1) $tr \models \mathbf{X}_c \varphi \iff tr \models \neg \mathbf{Y}_c \neg \varphi$,
- 2) $tr \models \mathbf{F}_L \varphi \iff tr \models \neg \mathbf{G}_L \neg \varphi$.

As usual, other Boolean connectives like $\vee, \rightarrow, \equiv$ may be defined by means of \neg and \wedge . Some other CTL^* operators like, for example, \mathbf{R} (release) or \mathbf{W} (weak until) may be parametrized by environmental behaviour patterns in the same fashion.

The model checking problem we deal with is that of checking, given a finite state transducer Π operating over a semigroup (S, \circ, e) , and an $\mathcal{LP}\text{-CTL}^*$ formula φ , whether $TS(\Pi, S), d_{init} \models \varphi$ holds. When a semigroup is fixed then we use a brief notation $\Pi \models \varphi$.

IV. MODEL CHECKING AGAINST $\mathcal{LP}\text{-CTL}^*$ SPECIFICATIONS

In this paper we discuss only the most simple case of model checking problem for finite state transducers against $\mathcal{LP}\text{-CTL}^*$ formulae when

- the semigroup (S, \circ, e) the transducers operate over is a *free monoid*, which means that S is the set of all finite words in the alphabet \mathcal{A} , the binary operation \circ is concatenation of words, and the neutral element e is the empty word ε ;
- the family of environment behaviour patterns \mathcal{P} is the family of regular languages in the alphabet \mathcal{C} ;
- all basic predicates in \mathcal{P} are specified by regular languages in the alphabet \mathcal{A} .

All regular languages used as environment behaviour patterns and basic predicate specifications are defined by means of deterministic finite state automata (DFAs). Therefore, the size of a $\mathcal{LP}\text{-CTL}^*$ formula is the number of Boolean connectives and temporal operators occurred in φ plus the total size of automata used in φ to specify environment behaviour patterns and basic predicates.

Let us first describe a model checking algorithm for $\mathcal{LP}\text{-CTL}$ fragment of $\mathcal{LP}\text{-CTL}^*$, which consists of all $\mathcal{LP}\text{-CTL}^*$ formulae such that every temporal operator $\mathbf{X}_c, \mathbf{Y}_c, \mathbf{F}_L, \mathbf{G}_L, \mathbf{U}_L$ is immediately preceded by a trajectory quantifier \mathbf{E} or \mathbf{A} . In our algorithm we involve an explicit iterative model checking techniques for the ordinary CTL (see [10], [8]). Following this approach satisfiability checking of a formula φ in a state d of a model M is reduced to satisfiability checking of the largest subformulae of φ in the state d and in the neighboring states of M . In other words, a model checking procedure incrementally labels all states of a model by those subformulae of φ which are satisfied in these states.

Let $\Pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ be a finite state transducer over the free semigroup $(\mathcal{A}^*, \cdot, \varepsilon)$ and let φ be an $\mathcal{LP}\text{-CTL}$ formula. There are five pairs of coupled $\mathcal{LP}\text{-CTL}$ temporal operators: \mathbf{AX}_c and \mathbf{EX}_c , \mathbf{AY}_c and \mathbf{EY}_c , \mathbf{AF}_L and \mathbf{EF}_L ,

\mathbf{AG}_L and \mathbf{EG}_L , \mathbf{AU}_L and \mathbf{EU}_L . As in the case of “ordinary” CTL (see [10]), each of these couple can be expressed in terms of four main coupled operators $\mathbf{EX}_c, \mathbf{EY}_c, \mathbf{EG}_L$ and \mathbf{EU}_L :

Proposition 2. *For every formula φ the following equalities hold*

- 1) $\models \mathbf{AX}_c \varphi \equiv \neg \mathbf{EY}_c \neg \varphi$,
- 2) $\models \mathbf{AY}_c \varphi \equiv \neg \mathbf{EX}_c \neg \varphi$,
- 3) $\models \mathbf{AF}_L \varphi \equiv \neg \mathbf{EG}_L \neg \varphi$,
- 4) $\models \mathbf{EF}_L \varphi \equiv \mathbf{E}[\text{true } \mathbf{U}_L \varphi]$,
- 5) $\models \mathbf{AG}_L \varphi \equiv \neg \mathbf{EF}_L \neg \varphi$,
- 6) $\models \mathbf{A}[\varphi \mathbf{U}_L \psi] \equiv \neg \mathbf{E}[\neg \psi \mathbf{U}_L (\neg \varphi \wedge \neg \psi)] \wedge \neg \mathbf{EG}_L \neg \psi$.

Certainly, some other relationships like fixed-point identities are also valid in $\mathcal{LP}\text{-CTL}^*$ (see [17]) but they will not be involved in this paper for model checking purpose.

We can now bound our consideration with those $\mathcal{LP}\text{-CTL}$ formulae which are constructed using only $\neg, \wedge, \mathbf{EX}_c, \mathbf{EY}_c, \mathbf{EG}_L$ and \mathbf{EU}_L . Let M be a transition system $TS(\Pi, \mathcal{A}^*) = (D, \mathcal{C}, d_{init}, T)$ of Π over \mathcal{A}^* . It should be noticed that M is, in general, infinite. So, in order to obtain an effective model checking procedure we need a construction that will model the behaviour of M with respect to a target formula φ .

For every basic predicate $P \in \mathcal{P}$ let $A_P = (Q_P, \mathcal{A}, \text{init}_P, \delta_P, F_P)$ be a *minimal* DFA recognizing this language. Here Q_P is a finite set of states, init_P is an initial state, F_P is a set of accepting states and $\delta_P: Q_P \times \mathcal{A} \rightarrow Q_P$ is a transition function. The latter can be extended to the set \mathcal{A}^* in the usual fashion:

$$\delta_P(q_P, \varepsilon) = q_P \quad \text{and} \quad \delta_P(q_P, \gamma a) = \delta_P(\delta_P(q_P, \gamma), a).$$

Let P_1, P_2, \dots, P_k be all basic predicates occurred in the formula φ . Given a transducer $\Pi = (Q, \mathcal{C}, \mathcal{A}, q_{init}, T)$ and a formula φ , we build a *checking machine* — a transducer $\mathcal{M} = (\hat{Q}, \mathcal{C}, \mathcal{A}, \hat{q}_{init}, \hat{T})$, where

- $\hat{Q} = Q \times Q_{P_1} \times \dots \times Q_{P_k}$ is a set of states (to avoid misunderstanding we will call them metastates);
- $\hat{q}_{init} = (q_{init}, \text{init}_{P_1}, \dots, \text{init}_{P_k})$ is an initial metastate;
- $\hat{T} \subseteq \hat{Q} \times \mathcal{C} \times \hat{Q} \times \mathcal{A}^*$ is a transition relation, such that:

$$(\hat{q}', c, \hat{q}'', h) \in \hat{T} \iff \begin{cases} (q'_\pi, c, q''_\pi, h) \in T \text{ and} \\ \delta_{P_j}(q'_j, h) = q''_j \\ \text{for all } j, 1 \leq j \leq k. \end{cases}$$

Thus, every metastate is a tuple $\hat{q} = (q_0, q_1, \dots, q_k)$ such that $q_0 \in Q$ and $q_j \in Q_{P_j}$ for every $j, 1 \leq j \leq k$, and the transition relation \hat{T} synchronizes transitions of Π and the automata A_{P_1}, \dots, A_{P_k} in response to every signal c . Recall that the elements of the free monoid are words s from \mathcal{A}^* . The checking machine \mathcal{M} induces a binary relation \sim on the set D : for an arbitrary pair $d' = (q', s')$ and $d'' = (q'', s'')$ of states of computation of Π over \mathcal{A}^*

$$d' \sim d'' \iff \begin{cases} q' = q'' \text{ and} \\ \delta_{P_j}(\text{init}_{P_j}, s') = \delta_{P_j}(\text{init}_{P_j}, s'') \text{ for all } j. \end{cases}$$

The relation \sim is clearly an equivalence relation of finite index, and every equivalence class of states of computation in M corresponds to a metastate of the checking machine \mathcal{M} . As it can be seen from the definition of \sim , if two states of computation d' and d'' are equivalent and there is a trajectory $tr' = (d', \alpha')$ in M , where $\alpha' = (c_1, d'_1), (c_2, d'_2), \dots$, from one of these states, then there is also a corresponding trajectory $tr'' = (d'', \alpha'')$, where $\alpha'' = (c_1, d''_1), (c_2, d''_2), \dots$ from the other state, such that $d'_i \sim d''_i$ holds for every $i, i \geq 1$. Actually, this means that \sim is a bisimulation relation on the state space of the transition system M . It is well known (see [3], [8]) that bisimulation preserves the satisfiability of CTL formulae. The Proposition below shows that the same is true for $\mathcal{LP}\text{-}CTL$. This means that the checking machine provides a finite contraction of the infinite transition system $M = TS(\Pi, \mathcal{A}^*)$ w.r.t. satisfiability of $\mathcal{LP}\text{-}CTL$ formulae.

Proposition 3. *Suppose that d' and d'' are two states of computation in M such that $d' \sim d''$. Then $M, d' \models \varphi \iff M, d'' \models \varphi$.*

Proof. It is carried out by induction on the nesting depth of φ . When φ is a basic predicate the assertion obviously follows from the definition of \sim . Equally obvious are the cases when $\varphi = \neg\psi$ and $\varphi = \psi_1 \wedge \psi_2$. We focus only on the case of $\varphi = \mathbf{E}[\psi \mathbf{U}_L \chi]$; the other cases when φ is of the form $\mathbf{EX}_c\psi$, $\mathbf{EY}_c\psi$, or $\mathbf{EG}_L\psi$ can be treated similarly.

Suppose that $M, d' \models \mathbf{E}[\psi \mathbf{U}_L \chi]$. Then, by the definition of $\mathcal{LP}\text{-}CTL$ semantics, there exists a trajectory $tr' = (d', \alpha')$, such that $M, tr' \models \psi \mathbf{U}_L \chi$ and $\alpha' = (c_1, d'_1), (c_2, d'_2), \dots$. As it was noticed above, there is also a corresponding trajectory $tr'' = (d'', \alpha'')$ in M , where $\alpha'' = (c_1, d''_1), (c_2, d''_2), \dots$, such that $d'_i \sim d''_i$ holds for every $i, i \geq 1$. Then, by induction hypotheses, $M, d'_i \models \psi \iff M, d''_i \models \psi$ and $M, d'_i \models \chi \iff M, d''_i \models \chi$ hold for every $i, i \geq 1$.

Since $M, tr' \models \psi \mathbf{U}_L \chi$, there exists i such that

- 1) $c_1 c_2 \dots c_i \in L$ and $M, tr'|^i \models \chi$;
- 2) for all $j < i$ if $c_1 c_2 \dots c_j \in L$ then $M, tr'|^j \models \psi$.

But, taking into account the fact that ψ and χ are state formulas, we must recognize that $M, tr''|^i \models \chi$ and that $M, tr''|^j \models \psi$ every time when $M, tr'|^j \models \psi$. Thus, we arrive at the conclusion that $M, tr'' \models \psi \mathbf{U}_L \chi$ and, hence, $M, d'' \models \mathbf{E}[\psi \mathbf{U}_L \chi]$. \square

Each metastate $\hat{q} = (q_0, q_1, \dots, q_k)$ of the checking machine \mathcal{M} represents an equivalence class $D_{\hat{q}}$ which includes all states $d = (q, h) \in D$ such that $q = q_0$ and $\delta_{P_j}(\text{init}_{P_j}, h) = q_j$ for all $j, 1 \leq j \leq k$. Taking into account Proposition 3, we can correctly introduce a new satisfiability relation \models_0 on the metastates of the checking machine:

$$\hat{q} \models_0 \varphi \iff \text{for some } d \in D_{\hat{q}}: M, d \models \varphi.$$

Not only the states of the transition system $M = TS(\Pi, S)$ correspond to the metastates of the checking machine \mathcal{M} , but also there is a relationship between the trajectories in M and the traces in \mathcal{M} (they can be quite naturally called metatrajectories). More formally, every trajectory $tr = (d_0, \alpha)$ in M

with $\alpha = (c_1, d_1)(c_2, d_2) \dots$, corresponds to a *metatrajectory* $\hat{tr} = (\hat{q}_0, \hat{\alpha})$, where $\hat{\alpha} = (c_1, \hat{q}_1)(c_2, \hat{q}_2) \dots$ is such that for all $i \geq 0$: $d_i \in D_{\hat{q}_i}$. It is easy to see that every metatrajectory $\hat{tr} = (\hat{q}_0, \hat{\alpha})$ corresponds to the only trajectory $tr = (d_0, \alpha)$ which originates in a given state d_0 from $D_{\hat{q}_0}$.

The well-known labeling algorithm for conventional CTL and ordinary Kripke structures can be now adapted in such a way as to cope with model checking problem for $\mathcal{LP}\text{-}CTL$. The algorithm operates as follows. For every metastate $\hat{q} \in \hat{Q}$ of the checking machine \mathcal{M} it computes a set $label(\hat{q})$ of all subformulae of φ which are satisfied in \hat{q} . More formally, let $Sub(\varphi)$ be the minimal set of $\mathcal{LP}\text{-}CTL$ formulae such that:

- 1) $\varphi \in Sub(\varphi)$;
- 2) if $\neg\psi \in Sub(\varphi)$ then $\psi \in Sub(\varphi)$;
- 3) if $\psi \wedge \chi \in Sub(\varphi)$ then $\psi, \chi \in Sub(\varphi)$;
- 4) if $\mathbf{EX}_c\psi \in Sub(\varphi)$, $\mathbf{EY}_c\psi \in Sub(\varphi)$ or $\mathbf{EG}_L\psi \in Sub(\varphi)$ then $\psi \in Sub(\varphi)$;
- 5) if $\mathbf{E}[\psi \mathbf{U}_L \chi] \in Sub(\varphi)$ then $\psi, \chi \in Sub(\varphi)$.

The algorithm builds incrementally the sets $label(\hat{q})$ of all those $\psi \in Sub(\varphi)$ for which $\hat{q} \models_0 \psi$ holds. At the first step every $label(\hat{q})$ contains only basic predicates, i. e. $label(\hat{q}) \subseteq Sub(\varphi) \cap \mathcal{P}$. Then, at *step* i the algorithm processes those subformulae ψ whose nesting depth is $i - 1$. Every time when the algorithm adds a subformula ψ to $label(\hat{q})$ it thus detects that $\hat{q} \models_0 \psi$.

All we need now is to describe how the algorithm should process formulae of 7 types: basic predicate P , $\neg\psi$, $\psi_1 \wedge \psi_2$, $\mathbf{EX}_c\psi$, $\mathbf{EY}_c\psi$, $\mathbf{EG}_L\psi$ and $\mathbf{E}[\psi \mathbf{U}_L \chi]$.

- A basic predicate P_i is added to $label(\hat{q})$ iff $\hat{q} = (q_0, q_1, \dots, q_i, \dots, q_k)$ and $\hat{q}_i \in F_{P_i}$, $i \geq 1$;
- A subformula $\neg\psi$ is added to $label(\hat{q})$ iff $\psi \notin label(\hat{q})$;
- A subformula $\psi_1 \wedge \psi_2$ is added to $label(\hat{q})$ iff both $\psi_1, \psi_2 \in label(\hat{q})$;
- A subformula $\mathbf{EX}_c\psi$ is added to $label(\hat{q})$ iff there exists a transition $\hat{q} \xrightarrow{c, h} \hat{q}'$ such that $\psi \in label(\hat{q}')$;
- A subformula $\mathbf{EY}_c\psi$ is added to $label(\hat{q})$ iff there exists a transition $\hat{q} \xrightarrow{c, h} \hat{q}'$ such that $\psi \in label(\hat{q}')$ or there exists a transition $\hat{q} \xrightarrow{c', h} \hat{q}'$ such that $c' \neq c$;
- To handle a subformula $\mathbf{E}[\psi \mathbf{U}_L \chi]$ we construct a directed labeled graph (DLG) $\Gamma_U(\mathcal{M}, L)$ as follows. Let $A_L = (Q_L, \mathcal{C}, \text{init}_L, \delta_L, F_L)$ be a minimal DFA that recognizes the language L . Then the nodes of $\Gamma_U(\mathcal{M}, L)$ are all pairs $(\hat{q}, q_L) \in \hat{Q} \times Q_L$. This DLG has an arc of the form $(\hat{q}', q'_L) \xrightarrow{c, h} (\hat{q}'', q''_L)$ iff $\hat{q}' \xrightarrow{c, h} \hat{q}''$ is a transition of \mathcal{M} and $\delta_L(q'_L, c) = q''_L$.

We then delete all those nodes (\hat{q}, q_L) of $\Gamma_U(\mathcal{M}, L)$ for which the relations $\psi \notin label(\hat{q})$, $\chi \notin label(\hat{q})$ and $q_L \in F_L$ hold simultaneously and discard all arcs that income to or outcome from such nodes. A DLG thus reduced is denoted by $\Gamma'_U(\mathcal{M}, L)$.

A subformula $\mathbf{E}[\psi \mathbf{U}_L \chi]$ is added to the set $label(\hat{q})$ iff $\Gamma'_U(\mathcal{M}, L)$ includes the node (\hat{q}, init_L) and there exists a directed path in this graph from this node to some node (\hat{q}', q'_L) such that $\chi \in label(\hat{q}')$ and $q'_L \in F_L$.

- For a subformula $\mathbf{EG}_L\psi$ we construct a DLG $\Gamma_G(\mathcal{M}, L)$ in the same fashion and delete all the nodes (\hat{q}, q_L) for which the relations $\psi \notin \text{label}(\hat{q})$ and $q_L \in F_L$ hold simultaneously. As the result we obtain the reduced DLG $\Gamma'_G(\mathcal{M}, L)$.

The subformula $\mathbf{EG}_L\psi$ is added to the set $\text{label}(\hat{q})$ iff $\Gamma'_G(\mathcal{M}, L)$ includes the node (\hat{q}, init_L) and there exists a directed path in this graph from this node to some nontrivial *strongly connected component* (SCC), that is, to a subgraph, every node of which is reachable from itself by some non-empty path.

As soon as all the subformulae from $\text{Sub}(\varphi)$ (including the formula φ) are processed we obtain the result of the model checking as

$$\Pi \models \varphi \iff \varphi \in \text{label}(\hat{q}_{\text{init}}).$$

The correctness of this assertion is based on the following relationship: $\hat{q} \models_0 \varphi \iff \varphi \in \text{label}(\hat{q})$. It can be proved by applying induction on the nesting depth of formulae with the help of Proposition 3. We also need Propositions 4 and 5 to justify the induction step for formulae of the form $\mathbf{E}[\psi \mathbf{U}_L \chi]$ and $\mathbf{EG}_L\psi$.

Suppose, that for every metastate $\hat{q} \in \hat{Q}$ it is true that $\hat{q} \models_0 \psi \iff \psi \in \text{label}(\hat{q})$ and $\hat{q} \models_0 \chi \iff \chi \in \text{label}(\hat{q})$. This statement is used as an inductive hypothesis.

Proposition 4. *Let $\hat{q}_0 \in \hat{Q}$ be an arbitrary metastate in \mathcal{M} . Then $\hat{q}_0 \models_0 \mathbf{E}[\psi \mathbf{U}_L \chi]$ iff some node (\hat{q}', q'_L) in DLG $\Gamma'_U(\mathcal{M}, L)$ such that $\hat{q}' \models_0 \chi$ and $q'_L \in F_L$ is reachable from the node $(\hat{q}_0, \text{init}_L)$ by a directed path.*

Proposition 5. *Let $\hat{q}_0 \in \hat{Q}$ be an arbitrary metastate in \mathcal{M} . Then $\hat{q}_0 \models_0 \mathbf{EG}_L\psi$ iff some nontrivial strongly connected component is reachable from the node $(\hat{q}_0, \text{init}_L)$ in DLG $\Gamma'_G(\mathcal{M}, L)$ by a directed path.*

The proofs of these Propositions are straightforward adaptations of the correctness proof of the tabular model checking algorithm for *CTL* which is discussed in much details in [8]. However, for completeness of the exposition we give here a proof of Proposition 5. The proof of Proposition 4 follows the similar line of reasoning.

Proof of Proposition 5. (Sketch)

(\Rightarrow) Suppose, that $\hat{q}_0 \models_0 \mathbf{EG}_L\psi$. Consider an arbitrary state $d_0 \in D_{\hat{q}_0}$. Then, by definition of \models_0 and by Proposition 3, it is true that $M, d_0 \models \mathbf{EG}_L\psi$. This means that there is a trajectory $tr = (d_0, \alpha)$, where $\alpha = (c_1, d_1), (c_2, d_2), \dots$, such that $M, tr \models \mathbf{G}_L\psi$. By the semantics of $\mathcal{LP}\text{-CTL}^*$, $M, d_i \models \psi$ holds for every i such that $c_1 c_2 \dots c_i \in L$.

Consider now the corresponding metatrayjectory $\hat{tr} = (\hat{q}_0, \hat{\alpha})$ in the checking machine, where $\hat{\alpha} = (c_1, \hat{q}_1), (c_2, \hat{q}_2), \dots$, and let

$$\pi = (\hat{q}_0, \text{init}_L) \xrightarrow{c_1, h_1} (\hat{q}_1, q_{1L}) \xrightarrow{c_2, h_2} (\hat{q}_2, q_{2L}) \xrightarrow{c_3, h_3} \dots,$$

be the respective path in the DLG $\Gamma_G(\mathcal{M}, L)$ which originates in the node $(\hat{q}_0, \text{init}_L)$. Relying on Proposition 3 and taking

into account the fact that $q_{iL} = \delta_L(\text{init}_L, c_1 c_2 \dots c_i)$ for every $i, i \geq 0$, we may conclude that $\hat{q}_i \models_0 \psi$ holds for every i such that $q_{iL} \in F$. By induction hypothesis, $\hat{q}_i \models_0 \psi$ is equivalent to $\psi \in \text{label}(\hat{q}_i)$. Therefore, by definition of DLG $\Gamma_G(\mathcal{M}, L)$ the path π is the infinite path which is entirely contained in the $\Gamma'_G(\mathcal{M}, L)$. Due to the finiteness of $\Gamma'_G(\mathcal{M}, L)$, this path may be represented as a concatenation $\pi = \pi_1 \pi_2$, where π_1 is a finite path, and π_2 is an infinite path passing through each of its nodes infinitely often. It is clear that the set $V(\pi_2)$ of all nodes of π_2 is included in some strongly connected component. Thus, a nontrivial strongly connected component is reachable from the node $(\hat{q}_0, \text{init}_L)$ in DLG $\Gamma'_G(\mathcal{M}, L)$.

(\Leftarrow) Suppose, that a nontrivial strongly connected component is reachable from the node $(\hat{q}_0, \text{init}_L)$ in DLG $\Gamma'_G(\mathcal{M}, L)$. Then there exists an infinite path

$$\pi = (\hat{q}_0, \text{init}_L) \xrightarrow{c_1, h_1} (\hat{q}_1, q_{1L}) \xrightarrow{c_2, h_2} (\hat{q}_2, q_{2L}) \xrightarrow{c_3, h_3} \dots$$

in $\Gamma'_G(\mathcal{M}, L)$ from the node $(\hat{q}_0, \text{init}_L)$. Consider now the sequence of the first components \hat{q}_i of all nodes $(\hat{q}_i, q_{iL}), i \geq 0$, occurred in this path. By the definition of the DLG $\Gamma'_G(\mathcal{M}, L)$,

- 1) this sequence is a metatrayjectory \hat{tr} in the checking machine \mathcal{M} ,
- 2) $\psi \in \text{label}(\hat{q}_i)$ holds for every node (\hat{q}_i, q_{iL}) such that $q_{iL} \in F_L$.

By the induction hypothesis, the latter implies $\hat{q}_i \models_0 \psi$ for every metastate \hat{q}_i in this trajectory such that $c_1 c_2 \dots c_i \in L$. Consider an arbitrary state $d_0 \in D_{\hat{q}_0}$ and a trajectory $tr = (d_0, \alpha)$ in M , where $\alpha = (c_1, d_1), (c_2, d_2), \dots$, which corresponds to \hat{tr} . By definition of \models_0 and Proposition 3, $M, d_i \models \psi$ holds for every i such that $c_1 c_2 \dots c_i \in L$. Then, according to the semantics of $\mathcal{LP}\text{-CTL}^*$, $M, tr \models \mathbf{G}_L\psi$, and, hence, $M, d_0 \models \mathbf{EG}_L\psi$. Thus, by referring once again to definition of \models_0 , we arrive at the conclusion that $\hat{q}_0 \models_0 \mathbf{EG}_L\psi$. \square

Now we estimate the complexity of the model checking algorithm for $\mathcal{LP}\text{-CTL}$ described above. By the *size of a transducer* $\Pi = (Q, \mathcal{C}, \mathcal{A}, q_{\text{init}}, T)$ we will mean the sum $\|\Pi\| = |Q| + |T|$. The *size of a formula* φ is defined as follows. Suppose that basic predicates $\{P_i\}_{i=1}^k$ occurred in φ are recognized by minimal DFAs $\{A_{P_i} = (Q_{P_i}, \mathcal{A}, \text{init}_{P_i}, \delta_{P_i}, F_{P_i})\}_{i=1}^k$. Suppose also that environment patterns $\{L_i\}_{i=1}^s$ used in φ are recognized by minimal DFAs $\{A_{L_i} = (Q_{L_i}, \mathcal{A}, \text{init}_{L_i}, \delta_{L_i}, F_{L_i})\}_{i=1}^s$. Then the size of φ is the sum $\|\varphi\| = |\text{Sub}(\varphi)| + \sum_{i=1}^k |Q_{P_i}| + \sum_{i=1}^s |Q_{L_i}|$.

As it can be seen from the description of our model checking algorithm, the size of auxiliary graphs $\Gamma'_U(\mathcal{M}, L)$ and $\Gamma'_G(\mathcal{M}, L)$ used in this algorithm does not exceed the value $\|\Pi\| \cdot \left(\prod_{i=0}^k |Q_{P_i}| \right) \cdot \max(|Q_{L_i}| : 1 \leq i \leq s)$. These graphs are processed in no more than $|\text{Sub}(\varphi)|$ steps. So, the total time complexity of our model checking algorithm does not exceed the value $\|\Pi\| \cdot |\text{Sub}(\varphi)| \cdot \left(\prod_{i=0}^k |Q_{P_i}| \right) \cdot \max(|Q_{L_i}| : 1 \leq i \leq s)$ which is $O(\|\Pi\| \cdot 2^{\|\varphi\|})$.

As a result of these considerations, we get the following

Theorem 1. *Model checking of a finite state transducer Π operating over a free monoid against a formula $\varphi \in \mathcal{LP}\text{-CTL}$ can be performed in time $O(\|\Pi\| \cdot 2^{\|\varphi\|})$.*

When a more general case of model checking problem of FSTs against $\mathcal{LP}\text{-CTL}^*$ formulae is concerned we can rely on the well-known combining approach which is based on the interleaving application of model checking algorithms for CTL and LTL . The details can be found in [8]. The similar procedure for $\mathcal{LP}\text{-CTL}^*$ can be obtained in the same fashion by means of $\mathcal{LP}\text{-CTL}$ model checking algorithm described above and $\mathcal{LP}\text{-LTL}$ model checking algorithm developed in [17]. Since this approach does not take into account any specific features of $\mathcal{LP}\text{-CTL}^*$ formulae, we will not give a complete description of it.

V. $\mathcal{LP}\text{-CTL}^*$ AND ORDINARY KRIPKE STRUCTURES

In this section we consider the model checking problem for two subfamilies of $\mathcal{LP}\text{-CTL}^*$ whose semantics can be defined on ordinary Kripke structures.

Recall, that a *Kripke structure* over a finite set AP of atomic propositions is a quadruple $M = (Q, q_{init}, R, \rho)$, where Q is a finite set of states which includes an initial state q_{init} , $R \subseteq Q \times Q$ is a transition relation and $\rho : Q \rightarrow 2^{AP}$ is a *labeling function* which for each state q gives a matching set $\rho(q) \subseteq AP$ of all atomic propositions that are evaluated to *true* in this state. As usual, the *size* of M is the sum $\|M\| = |Q| + |R|$. Below we present two modifications of $\mathcal{LP}\text{-CTL}^*$ that are well suited for model checking of Kripke structures.

Given a Kripke structure $M = (Q, q_{init}, R, L)$, consider a set of $\mathcal{LP}\text{-CTL}^*$ formulae where \mathcal{L} is a family of regular languages over one-letter alphabet $\{c\}$ and $\mathcal{P} = AP$ (we denote this formulae by $\mathcal{LP}\text{-1-CTL}^*$) and a transition system $M_c = (Q, \{c\}, q_{init}, R_c, L)$ where $(q', c, q'') \in R_c$ iff $(q', q'') \in R$. Then for $q \in Q$ the relation $q \models P$ holds iff $P \in \rho(q)$. The semantics of more complex formulae is defined exactly as in Section III.

Some $\mathcal{LP}\text{-1-CTL}^*$ formulae have an ability to keep track of the number of steps of the run. For example, an $\mathcal{LP}\text{-1-LTL}$ formula $\mathbf{AG}_L \varphi$, where $L = \{c^{2n}\}$ is a regular language which contains all 1-letter words of even length, expresses the assertion that φ holds at every even step of a run. By using the techniques of Ehrenfeucht-Fraisse games for Temporal Logics developed and studied in [11] one can prove that this property can not be specified by means of usual LTL . This certifies that $\mathcal{LP}\text{-1-CTL}^*$ is more expressive than CTL^* and justifies its use as a new specification language for finite state transducers and Kripke structures.

Observe, that given a set AP of all atomic propositions used in formulae we can use the M_c directly as a checking machine \mathcal{M} for the algorithm described in Section IV. Suppose that formula φ refers to 1-letter regular languages L_1, L_2, \dots, L_s as the parameters of temporal operators, and every language L_i , $1 \leq i \leq s$, is recognized by a DFA with a set of states Q_{L_i} . Then the size of the graphs used in this algorithm does not exceed the value $\|M\| \cdot \max(|Q_{L_i}| : 1 \leq i \leq s)$ which is $O(\|M\| \cdot \|\varphi\|)$, where $\|\varphi\| = |Sub(\varphi)| + \sum_{i=0}^s |Q_{L_i}|$.

Another modification of the Kripke structure M allows to encode more detailed information of the computation flow. Consider the alphabet $\Sigma = 2^{AP}$. For each state q in M there exists a letter $\sigma_{\rho(q)} \in \Sigma$ corresponding to the label $\rho(q)$ assigned to this state.

Let $M_{AP} = (Q \cup \{err\}, q_{init}, R_{AP}, \rho_{AP})$ be a transition system for M , where for every $q \in Q$ the following equalities hold: $\rho_{AP}(q) = \rho(q)$, $\rho_{AP}(err) = \{err\}$ and $R_{AP} \subseteq Q \times 2^{AP} \times Q$ is a minimal transition relation such that:

- for each transition (q', q'') of the Kripke structure M there exists a *fair transition* $(q', \sigma_{\rho(q'')}, q'')$ and *erroneous transitions* (q', σ, err) for each $\sigma \neq \sigma_{\rho(q'')}$;
- $(err, \sigma, err) \in R_{AP}$ holds for each $\sigma \in \Sigma$ and $(err, \sigma, q) \notin R_{AP}$ holds for each $q \neq err$.

Then consider a specification language $\mathcal{LP}\text{-n-CTL}^*$ which is a set of all such formulae where \mathcal{L} is a family of regular languages over Σ and $\mathcal{P} = AP$. To model check a transition system M_{AP} against these formulae one needs to process only the states in Q and only the fair transitions. To do so, we replace all state formulae of type $\mathbf{A}\varphi$ with $\mathbf{A}(\mathbf{G} \neg err \rightarrow \varphi)$ and all state formulae of type $\mathbf{E}\varphi$ with $\mathbf{E}(\mathbf{G} \neg err \wedge \varphi)$. The transition system M_{AP} thus obtained may as well be used as a checking machine for the model checking algorithm described in Section IV.

Thereby, the following theorem holds.

Theorem 2.

- 1) *There exists an algorithm for model checking of a Kripke structure M against a formula $\varphi \in \mathcal{LP}\text{-1-CTL}$ with time complexity $O(\|M\| \cdot \|\varphi\|^2)$.*
- 2) *There exists an algorithm for model checking of a Kripke structure M against a formula $\varphi \in \mathcal{LP}\text{-n-CTL}$ with time complexity $O(\|M\| \cdot \|\varphi\|^2 \cdot 2^{|AP|})$.*

As it can be seen from this theorem, the exponential complexity of model checking procedure described in Section IV is due to the language-theoretic nature of basic predicates used in $\mathcal{LP}\text{-CTL}^*$.

VI. RELATED PAPERS AND CONCLUSION

Actually, the idea of providing parametrization of temporal operators is not new. In [27] right-linear grammar patterns were offered to define new temporal operators. The same kind of temporal patterns but specified by means of finite state automata were introduced in [18], [24]. For these extensions it was proved that they have the same expressiveness as S1S and that satisfiability checking problem in these logics is PSPACE-complete. We did not pursue a goal of merely expanding the expressive possibilities of CTL^* ; our aim was to make CTL^* more adequate for describing the behaviour of reactive systems. Almost the same kind of parametrization is used in Dynamic LTL [14]. But our extension of CTL^* differs from that which was developed in [14], since in our logic basic predicates are also parameterized.

The $\mathcal{LP}\text{-CTL}^*$ formulae allows one to specify and verify the behaviour of finite state transducers that operate over

semigroups as well as classical Kripke structures. Moreover, when Kripke structures are concerned $\mathcal{LP}\text{-CTL}^*$ has more expressive power than conventional temporal logics. But the place of $\mathcal{LP}\text{-CTL}^*$ in the expressive hierarchy of specification languages, such as SIS, PDL or μ -calculus, has not yet been established and remains a matter for our further research.

The results of this paper combined with the results of [17] provide positive solution to model checking for transducers over free semigroups. Free semigroups is the most simple algebraic structure which can be used for interpretation of basic actions performed by transducers when they are regarded as formal models of sequential reactive systems. Next we are going to find out whether model checking algorithms could be built for transducers operating over more specific semigroups. Some preliminary results showed that this is not an easy problem. In [12] we proved that it is undecidable for the case of Abelian groups and free commutative semigroups. There still remains a bunch of other interesting semigroups classes (for example, different kinds of partially commutative semigroups) for which this problem remains open.

It is also interesting how much the complexity of model checking algorithms for $\mathcal{LP}\text{-CTL}^*$ depends on languages that are used as parameters of temporal operators. We assume that model checking problem becomes undecidable when context-free languages are allowed for this purpose.

The complexity issues of model checking for regular variant of $\mathcal{LP}\text{-CTL}^*$ also need further research. We assume that even for regular $\mathcal{LP}\text{-CTL}$ this problem is PSPACE-complete.

As for practical application of the results obtained in this article, the most important of them is the question of how easily it is possible to adapt the existing means of working with finite automata to widely known model checking tools (like SPIN, ν -SMV, etc.) in order to be able to effectively implement the proposed model checking algorithms for $\mathcal{LP}\text{-CTL}^*$.

The authors of the article thank the anonymous reviewers for their valuable comments and advice on improving the article.

This work was supported by the Russian Foundation for Basic Research, Grant N 18-01-00854.

REFERENCES

- [1] R. Alur, P. Cerny. *Streaming transducers for algorithmic verification of single-pass list-processing programs*. Proceedings of 38-th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, p. 599-610, 2011.
- [2] Alur R., Moarref S., and Topcu U.: Pattern-based refinement of assume-guarantee specifications in reactive synthesis. Proc. of 21-st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2015.
- [3] Baier C., Katoen J. Principles of Model Checking, 2008, MIT Press,
- [4] M. Blattner, T. Head. *The decidability of equivalence for deterministic finite transducers*. Journal of Computer and System Sciences, v. 1, p. 45-49, 1979.
- [5] M. Blattner, T. Head. *Single-valued a-transducers*. Journal of Computer and System Sciences, v. 15, p. 310-327, 1977.
- [6] K. Culik, J. Karhumaki. *The equivalence of finite-valued transducers (on HDTOL languages) is decidable*. Theoretical Computer Science, v. 47, p. 71-84, 1986.
- [7] Bouajjani A., Jonsson B., Nilsson M., Touili T.: Regular Model Checking. Proc. of 12-th International Conference on Computer Aided Verification, LNCS 1855 (2000), p. 403-418.
- [8] E. M. Clarke, Jr., O. Gramberg, D. A. Peled. *Model Checking*. MIT Press, 1999.
- [9] V. Diekert, G. Rozenberg eds. *The Book of Traces*, 1995, World Scientific, Singapore.
- [10] E.A. Emerson, J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. Journal of Computer and System Sciences, 1985, v. 30, N 1, p. 124.
- [11] K. Etessami, T. Wilke. *An Until Hierarchy and Other Applications of an Ehrenfeucht-Fraisse Game for Temporal Logic*. Information and Computation, v. 160, p. 88-108. Elsevier, 2000.
- [12] A. Gnatenko, V. A. Zakharov. *On the complexity of verification of finite state machines over commutative semigroups*. Proceedings of the 18-th International Conference "Problems of Theoretical Cybernetics" (Penza, June 20-24, 2017), p. 68-70.
- [13] T. Griffiths. *The unsolvability of the equivalence problem for free nondeterministic generalized machines*. Journal of the ACM 15, p. 409-413, 1968.
- [14] J.G. Henriksen, P.S. Thiagarajan. *Dynamic linear time temporal logic*. Annals of Pure and Applied Logic, 1999, v. 96, p. 187-207
- [15] Hu Q., D'Antoni L. Automatic Program Inversion using Symbolic Transducers. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, p. 376-389.
- [16] O. Ibarra *The unsolvability of the equivalence problem for Efree NGSMs with unary input (output) alphabet and applications*. SIAM Journal on Computing, v. 4, 1978.
- [17] D. G. Kozlova, V. A. Zakharov. *On the model checking of sequential reactive systems*. Proceedings of the 25th International Workshop on Concurrency, Specification and Programming (CS&P 2016), CEUR Workshop Proceedings, vol. 1698, p. 233-244. Humboldt Universitet Zu Berlin, 2016.
- [18] O. Kupferman, N. Piterman, M.Y. Vardi. *Extended Temporal Logic Revisited*. Proceedings of 12-th International Conference on Concurrency Theory, 2001, p. 519-535.
- [19] M. P. Schutzenberger. *Sur les relations rationnelles*. Proceedings of Conference on Automata Theory and Formal Languages, p. 209-213, 1975.
- [20] J. Sakarovitch, R. de Souza. *On the decomposition of k-valued rational relations*. Proceedings of 25-th International Symposium on Theoretical Aspects of Computer Science, p.621-632, 2008.
- [21] J. Sakarovitch, R. de Souza. *On the decidability of bounded valuedness for transducers*. Proceedings of the 33-rd International Symposium on MFCS, p. 588-600, 2008.
- [22] R. de Souza. *On the decidability of the equivalence for k-valued transducers*. Proceedings of 12-th International Conference on Developments in Language Theory, p. 252-263, 2008.
- [23] J. Thakkar, A. Kanade, R. Alur. *A transducer-based algorithmic verification of retransmission protocols over noisy channels*. Proceedings of IFIP Joint International Conference on Formal Techniques for Distributed Systems, p. 209-224, LNCS 7892, 2013.
- [24] M.Y. Vardi, P. Wolper. *Yet Another Process Logic* (Preliminary Version). Logic of Programs, 1983, p. 501-512.
- [25] M. Veanes, P. Hooimeijer, B. Livshits et al. *Symbolic finite state transducers: algorithms and applications*. Proceedings of the 39-th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM SIGPLAN Notices, v. 147, p. 137-150, 2012.
- [26] A. Weber. *Decomposing finite-valued transducers and deciding their equivalence*. SIAM Journal on Computing. v. 22, p. 175-202, 1993.
- [27] P. Wolper. *Temporal Logic Can Be More Expressive*. Information and Control, 1983, v. 56, N 1/2, p. 72-99.
- [28] Wolper P., Boigelot B. *Verifying systems with infinite but regular state spaces*. Proceedings of the 10-th Int. Conf. on Computer Aided Verification (CAV-1998). LNCS. 1427 (1998), p. 88-97.
- [29] V. A. Zakharov. *Equivalence checking problem for finite state transducers over semigroups*. Proceedings of the 6-th International Conference on Algebraic Informatics (CAI-2015), p. 208-221, LNCS 9270, 2015.

Tolerant parsing with a special kind of "Any" symbol: the algorithm and practical application

Alexey Goloveshkin

I. I. Vorovich Institute for Mathematics,
Mechanics and Computer Science
Southern Federal University
Milchakova str. 8a, 344090, Rostov-on-Don, Russia
Email: alexeyvale@gmail.com

Stanislav Mikhalkovich

I. I. Vorovich Institute for Mathematics,
Mechanics and Computer Science
Southern Federal University
Milchakova str. 8a, 344090, Rostov-on-Don, Russia
Email: miks@sfedu.ru

Abstract—Tolerant parsing is a form of syntax analysis aimed at capturing the structure of certain points of interest presented in a source code. While these points should be well-described in the corresponding language grammar, other parts of the program are allowed to be not presented in the grammar or to be described coarse-grained, thereby parser remains tolerant to the possible inconsistencies in the irrelevant area. Island grammars are one of the basic tolerant parsing techniques. "Island" is used as the relevant code alias, while the irrelevant code is called "water".

In the paper, a modified LL(1) parsing algorithm with built-in "Any" symbol processing is described. The "Any" symbol matches implicitly defined token sequences. The use of the algorithm for island grammars allows one to reduce irrelevant code description as well as to simplify patterns for relevant code matching. Our "Any" implementation is more accurate and less restrictive in comparison with the closest analogues implemented in Coco/R and LightParse parser generators. It also has potentially lower overhead than the "bounded seas" concept implemented in PetitParser. As shown in the experimental section, the tolerant parser generated by the C# island grammar is proven to be applicable for large-scale software projects analysis.

Index Terms—tolerant parsing, robust parsing, lightweight parsing, partial parsing, island grammar, parser generation

I. INTRODUCTION

Tolerant parsing is a parsing technique differing from the detailed whole-language (so-called baseline) parsing needed to build a full-featured compiler for a certain programming language. The main feature of the approach is the ability to capture points of interest inside the program, while all the code that doesn't contain such points can be skipped with no or minimal analysis performed. From developer's perspective, this feature allows her to focus on the structure of the points of interest, providing a minimal description of the irrelevant area. Tolerant parsing is usually called *lightweight* because tolerant grammar tends to be much shorter than the baseline one.

There are several reasons for the tolerant parsing to be the most suitable option for the program analysis:

- **Language embedding:** Some program artifacts assume the usage of multiple languages in one source file. In yacc-like grammars describing the syntax-directed translation, actions performed on a parsing step are expressed in terms of a certain general-purpose language. This means that the parser developed to capture the grammar

structure must be tolerant to all the possible variations of these language snippets. A possible application of a tolerant grammar parser is described in [1]. A detailed description of the embedded language tolerant parsing is given in [2].

- **Full grammar inaccessibility:** Tolerant grammar imprints the developer's notion of what places inside the program are the most important in the context of the current task. Its structure and the mapping between the grammar entities and the language constructs are transparent to the programmer from the very beginning and can be further refined in accordance with the in-the-wild testing results. On the contrary, the baseline grammar usage requires a prior exploration and comprehension. This process is proved to be time-consuming [3] and can be impossible due to proprietary issues or manual baseline parser writing [4].
- **Domain-specific idioms:** In a certain project, some local domain-specific patterns can be applied [4]. They represent a high-level abstraction layer which is not presented in the language syntax and obviously is out of scope of the whole-language parser. Nevertheless, tolerant parsers can be strictly focused at these patterns, ignoring the underlying structure, that allows one, in particular, to perform the impact analysis [5].
- **Incorrect program processing:** Syntax errors can be handled by the whole-language parser with some sophisticated error recovery mechanisms [6]–[8]. These mechanisms are heuristic by the nature and do not guarantee the successful parsing resumption, as well as the preservation of the built parts of the parse tree. Tolerant parser is able to skip irrelevant error-containing areas. At the same time, tolerant parsing can be broken by the mismatch of the elements structuring the program (e.g. by the absence of a block closing bracket in C#). Specific error handling techniques allowing recovering from this category of errors are described for the *bridge grammars* [9], [10], a special kind of the island grammars.

The contributions of this paper are: 1) a modification of the standard LL(1) parsing algorithm aimed at *island grammars*

tolerant parsing paradigm and designed to simplify irrelevant code skipping by means of a special `Any` symbol, this symbol is used in a tolerant grammar to mark an irrelevant code without specifying its structure; 2) a compiler generator with a built-in tolerant grammar description language containing `Any` as a part of the standard syntax; 3) a lightweight grammar of the C# programming language for this generator; 4) an experimental evidence of the applicability of the generated tolerant C# parser for large-scale software projects analysis.

The remainder of the paper is organized as follows: a brief overview of the existing tolerant parsing techniques is provided in Section II, in Section III the main goals of the current research are listed, in Section IV we discuss related work and outline limitations of the closest analogues of our approach, in Section V the modification of the standard LL(1) parsing algorithm aimed at `Any` symbol processing is introduced. The tolerant grammar for the C# programming language is presented in Section VI, this section also includes a sufficient volume of experimental data obtained by applying the generated tolerant parser to a real-world software source code. In Section VII a brief summary of the theoretical and practical contribution of the paper is provided.

II. TOLERANT PARSING TECHNIQUES

Three basic tolerant parsing techniques considered in [2], [4], [5], [11]–[14] are fuzzy parsing, island grammars and skeleton grammars.

Fuzzy parsing is based on the notion of *anchors*, specific tokens that mark the beginning of the constructs of interest. The formal definition of a fuzzy parser is provided in [11], [12]. The grammar used by the fuzzy parser actually consists of a number of smaller grammars. Each of them has its own start symbol with a production rule starting with the anchor. The main concern of the fuzzy parsing technique is that parsing process is tightly coupled with anchor tokens and can be error-prone in case these tokens appear outside of the points of interest.

Skeleton grammar construction is described in [13]. The skeleton grammar partially shares its structure with the baseline grammar. Rules describing points of interest are complemented with baseline grammar rules needed to derive those points from the start symbol (this process is called *root completion*). After the root completion, special *default productions* are formulated for all the undefined nonterminal symbols appearing in the rules added. The key precondition making this process possible is the baseline grammar accessibility. As noticed in Section I, most often this is not the case, besides, baseline grammar comprehension is quite time-consuming and requires some additional effort.

Island grammars technique is in the focus of our research. We believe that the concept of an island grammar is not well-known, so we provide its formal definition in accordance to [4], [5], despite the fact that this definition is not further referenced:

Definition 1: Given a context-free grammar $G = (N, T, P, S)$, where N is a set of nonterminal symbols, T is a

set of terminal symbols, P is a set of production rules, $S \in N$ is a specified start symbol, and a set of constructs of interest $I \subset T^*$ such that $\forall i \in I, \exists \omega_1, \omega_2 \in T^*: \omega_1 i \omega_2 \in L(G)$, where $L(G)$ denotes the language generated by G . An *island grammar* $G_I = (N', T', P', S')$ for $L(G)$ has the following properties:

- 1) $L(G) \subset L(G_I)$;
- 2) $\forall i \in I, \exists n \in N': n \xrightarrow{*} i$ and $\exists \omega_1, \omega_2 \in T^*: \omega_1 i \omega_2 \notin L(G) \wedge \omega_1 i \omega_2 \in L(G_I)$;
- 3) $K(G) > K(G_I)$.

The first property means that G_I generates an extension of $L(G)$, the second means that syntax analyser for G_I recognises constructs of interest from I in at least one sentence that is not recognized by the parser for G . The third property introduces the function $K(G)$ denoting the grammar complexity [15]. Informally speaking, island grammar consists of detailed productions describing certain constructs of interest (the *islands*) and liberal productions that catch the remainder (the *water*).

Island productions form a set of *patterns* to be matched by the points of interest. However, patterns are not enough to overcome two important island grammars side effects called *false positives* and *false negatives* [12]. In case relevant code snippets look similar to the irrelevant ones, they can be confused by the parser, as a result, the irrelevant code will be recognized as the point of interest and some points will be missed. To minimize this mismatch, iterative refinement is needed for patterns description as well as for *anti-patterns* matching irrelevant code. Besides, indeterministic parsing techniques are used to resolve the ambiguities. GLR [16] and GLL [17] parsers are capable to apply multiple parse actions for the same token in case of a deterministic parsing conflict and continue parsing the program in both ways. They also provides additional options to more accurately specify the parser behaviour. For instance, ASF+SDF Meta Environment IDE used in [4], [5] is capable to generate GLR parsers taking into account `{avoid}` and `{prefer}` constructs, which specify derivation preferences, and `{reject}` construct denoting water rules [18]. However, indeterministic parsing algorithms have a number of disadvantages: they are hard to trace, may return multiple parse trees that need some extra processing, and in case the islands look very similar to the water, a parsing result can be extremely unpredictable even after the iterative grammar refinement.

III. PROBLEM STATEMENT

The key assumption of the current research is that tolerant parsing can be performed with a deterministic algorithm, while patterns and anti-patterns forming the tolerant grammar can be simplified and partially eliminated by making the algorithm capable to match and skip some token sequences which have no explicit definition in the grammar.

The key goals of the current research are:

- 1) to design an LL(1) parsing algorithm with built-in notion of a special `Any` grammar symbol that provides skipping of the token sequences that are not explicitly described in the grammar;

- 2) to develop a compiler generator with an integrated language for LL(1) grammars writing, supporting Any symbol usage and automatic syntax tree construction;
- 3) to implement a tolerant island grammar for the C# programming language in the format supported by the generator below; the grammar is supposed to contain water anti-patterns simplified with Any symbol;
- 4) to test parser's applicability to the analysis of large-scale software projects.

The developed tool is planned to be used for lightweight parsing of software projects and their further sustainable concern-based markup.

IV. RELATED WORK

A. Coco/R

The first tool with embedded capability to match tokens from sets which are not directly specified in grammar is the Coco/R recursive-descent parsers generator. According to the documentation [19, p. 14], a special symbol ANY, which *denotes any token that is not an alternative to that ANY symbol in the current production*, is predefined in generated parsers. There is no formal description of this symbol processing, nevertheless, from the Coco/R open source code it is clear that for a given grammar an individual set of admissible tokens is connected with each ANY entry. Initially all the sets consist of all the tokens defined in grammar, then at the parser generation stage the alternatives of ANY symbols are removed from the corresponding sets to make the situation when parser has to make a choice between ANY and explicitly specified token unambiguously solvable in favour of the explicit option. Further we will call these alternatives *rivals*, in order to avoid terminological confusion with the alternatives of the rule.

The major shortcoming of ANY implementation in Coco/R is that the latter principle (the principle of priority of the explicitly specified token) is both incomplete and excessively restrictive. As a result, there are grammars for which parsers generated by Coco/R do not parse some programs valid from the developer's point of view. Some examples of such Coco/R grammars are shown in Figure 1. Lower case is used for terminal symbols, {} denotes zero or more repetitions of bracketed elements.

Excessive restrictiveness manifests itself for the iteration {ANY}, for which the same set defines admissible tokens both for the first position in the sequence corresponding to the {ANY} and for the rest positions. For the grammar in Figure 1a, the set {b, c} corresponds to ANY. The token a is excluded from this set, that makes all the strings starting from a being matched by the first alternative with explicitly written a token in the beginning. However, this also leads to the fact that the string bad\$ is not recognized by the parser. Note that the first token of the input — token b — is enough to choose the right production for A nonterminal, and the next a token can not be treated as the beginning of the first alternative. Therefore, a could be added to the set of admissible tokens for the second and subsequent positions, and this does not lead to ambiguity.

| | | |
|------------------------------------|--|--|
| $A = a \ b \ c \mid \{ANY\} \ d ;$ | $A = a \ B \ c \mid b \ B \ d ;$ $B = e \ \{ANY\} ;$ | $A = B \ c \mid d \ e ;$ $B = \{ANY\} \ a ;$ |
| (a) | (b) | (c) |

Fig. 1: The grammars illustrating ANY implementation shortcomings in Coco/R

The lack of outer context analysis for nonterminal symbols leads to incompleteness of the constraints that are imposed on the ANY admissible tokens set. In Figure 1b, ANY has no rivals within the rule, so the set of admissible elements consists of all the tokens defined in the grammar. As a result, there is no string that the parser is capable to recognize. Once the {ANY} processing starts, the parser reaches the end of the input stream treating each token as a part of the sequence corresponding to {ANY}. Outer context analysis for nonterminal B shows that tokens c and d may appear after ANY iteration, hence they must be deleted from admissible tokens set and be matched explicitly. In Figure 1c, only a token is not valid at ANY position according to the Coco/R algorithm, as a result, all the strings starting from d provoke an error due to ambiguity. At the same time, looking at the rule for A one can reveal that token d is the rival for ANY, as it is explicitly declared as the beginning of the second alternative.

Note that static analysis of the external context can lead to the construction of an excessively restrictive set of admissible tokens. For the grammar in Figure 1b, the set built with regard to the B outer context is {a, b, e}. Terminals c and d are not valid at ANY position, since they are in FOLLOW(B) and the optional ANY is at the end of the alternative for this nonterminal symbol. At the same time, after choosing an alternative for A, the more precise information about what can follow {ANY} is already available. When the first alternative is selected, d may be returned to the admissible tokens set. In case the second alternative is chosen, token c is admissible. That is, with dynamic decision making at the parsing stage, the set of programs recognizable by the parser can be extended.

In the current paper, the symbol Any is described. Unlike the ANY symbol in Coco/R, it corresponds to the sequence of zero or more tokens, not a single token. In its implementation, all the shortcomings listed above are eliminated and the decision about the current token's admissibility at Any position is made at the parsing stage.

B. LightParse

The tool for lightweight LALR(1) parsers development called LightParse [20] also supports the use of Coco/R-like Any symbol. LightParse application is similar to what we plan to do: generated lightweight parsers are used for concern-oriented source code markup [21]. LightParse performs static construction of the sets of tokens allowed at Any position and inherits all the Coco/R ANY implementation limitations. Besides, LightParse grammar is not directly used to generate the parser. Instead, it is translated to the YACC-like format supported by the standard LALR(1) parser generator GPPG, then GPPG produces the parser. In the translated grammar,

every entry of `Any` symbol is presented as a nonterminal symbol with single-element alternatives, by an alternative for each of the admissible terminal symbols. To get the valid YACC-like grammar without the nonterminal outer context analysis, LightParse imposes additional restrictions on `Any` usage: this symbol is not permitted to be in the end of the alternative, except for the start symbol productions. The presence of the intermediate grammar processing stage leads to inconsistency between the source grammar vocabulary, which is used by the grammar developer too, and the terms used in messages issued by the GPPG generator when some parser generation errors appear. Our `Any` implementation does not assume additional grammar adaptations for making the grammar suitable for the standard parsing algorithm. Instead, the standard LL(1) algorithm is modified to integrate the notion of `Any` and make it possible to define admissible tokens dynamically at the parsing stage. This eliminates the limitations of LightParse `Any` symbol.

C. Bounded seas

In [22], an extension of the regular parsing algorithm for parsing expression grammars (PEG) is described. It is intended to automatically deduce anti-patterns for water which is supposed to be context-aware, i.e., specific for each particular island in the input. This approach named *bounded seas* is integrated in PetitParser framework which allows one to implement PEG-based parsers in Smalltalk. Bounded seas are intended to completely eliminate the need for water rules explicit description in island grammars. A rule element of the form `~island~` is treated as a triple before-water island after-water. The key property of water is that it *never consumes any input from the right context of the bounded sea* [22]. The right context can be derived statically from the grammar or dynamically from the parser state. To make the latter possible, a stack of invoked expressions is added to the original PEG definition. For the after-water entity, right context is set with an expression consisting of all the possible expressions that can directly follow after-water, separated with the ordered choice operator. Right context for the before-water consists of the island expression itself and the corresponding after-water boundary expression which both are ordered choice operands. Water expression succeeds when the corresponding right boundary expression succeeds.

Checking all the possible right expressions assumes backtracking, which leads to a sufficient time overhead. Since backtracking is a basic technique for PEG due to ordered choice operator presence, it is usually optimised with *packrat* parsing [23], which makes parsing time linearly dependent on the length of the program. However, this results in a significant increase in the amount of memory used. Despite the right context exploration complexity, bounded seas are not able to make a globally correct decision on when water skipping should be ended. It is outlined in [22] that expressions forming the sea boundary actually recognize only prefixes

of the possible boundaries, and boundaries form an LL(k) language where k depends on the particular situation.

The approach presented in the current paper has less overhead because it does not use backtracking at all. It performs a linear input processing and use the modified FIRST set building algorithm to find a boundary for `Any`. Though in [22] standard FIRST and FOLLOW sets from LL(1) parsing theory are named insufficient to recognize the boundary, it is demonstrated in Section VI that with proper formulation of anti-patterns, the use of a modified FIRST set is enough to successfully analyse large-scale software project sources. `Any` symbol is used instead of explicit description of some parts of patterns and anti-patterns, that makes the island grammar significantly shorter and simplifies the grammar development process.

V. "ANY" SYMBOL IMPLEMENTATION

We are mainly focused not on the individual islands capturing but on the extraction of the program hierarchical structure up to a certain level and tend to name the relevant code not *islands* but *land*, so the developed parser generator was named Land¹ (by coincidence, it is also an acronym of "language description"). Table-driven predictive LL(1) parsing algorithm [8, pp. 220–228] was selected as the simplest and most suitable for debugging option for water skipping integration

A. Formal definition of a simplified grammar

We introduce into the grammar the special terminal symbol `Any` to mark places where zero or more tokens from the irrelevant area can be matched. We denote by $\text{lhs}(p)$ and $\text{rhs}(p)$, respectively, the left and the right part of the production p . Notation $x \in \text{rhs}(p)$ for $x \in N \cup T$ means that $\text{rhs}(p) = \alpha_1 x \alpha_2$, where $\alpha_1 \in (N \cup T)^*$, $\alpha_2 \in (N \cup T)^*$. $\text{SYMBOLS}(\gamma)$ is used for the set of terminal symbols needed to compose all the ω : $\gamma \xrightarrow{*} \omega, \gamma \in (N \cup T)^*, \omega \in T^*$. Through the symbol `Any`, we formulate the concept of a simplified grammar.

Definition 2: Let $G = (N, T, P, S)$ be a context-free grammar, $\text{Any} \notin T$. *Simplified* with respect to G is the grammar $G_s = (N_s, T_s, P_s, S_s)$ defined as follows:

- 1) $S_s = S$;
- 2) $P_s = \{p \in f(P) \mid \text{lhs}(p) = S_s \vee \exists p' \in P_s: \text{lhs}(p) \in \text{rhs}(p')\}$, where $f: P \rightarrow \{p = A \rightarrow \alpha \mid A \in N, \alpha \in (N \cup T \cup \{\text{Any}\})^*\}$ is the mapping that satisfies the following criteria:
 - a) $\exists P' \subseteq P: P' = \{p \in P \mid f(p) \neq p\}, P' \neq \emptyset$,
 - b) $\forall p \in P \setminus P', f(p) = p$,
 - c) $\forall p \in P', \exists n \in \mathbb{N}: p$ is representable in the form $A \rightarrow \alpha_1 \gamma_1 \beta_1 \alpha_2 \gamma_2 \beta_2 \dots \alpha_n \gamma_n \beta_n$ and $f(p)$ is representable in the form $A \rightarrow \alpha_1 \text{Any} \beta_1 \alpha_2 \text{Any} \beta_2 \dots \alpha_n \text{Any} \beta_n$, where $\forall i \in [1..n], \alpha_i \gamma_i \beta_i \in (N \cup T)^*$, and $\forall i \in [1..n], \forall a \in \text{FOLLOW}(A), \text{SYMBOLS}(\gamma_i) \cap \text{FIRST}(\beta_i \alpha_{i+1} \gamma_{i+1} \beta_{i+1} \dots \alpha_n \gamma_n \beta_n a) = \emptyset$;

¹<https://github.com/alexeyvaley/SYRCOSE-2018>

- 3) $N_s = \{A \in N \mid \exists p \in P_s: \text{lhs}(p) = A\};$
- 4) $T_s = \{a \in T \mid \exists p \in P_s: a \in \text{rhs}(p)\} \cup \{\text{Any}\}.$

Intuitively, P_s contains productions for the start symbol of G_s and productions for all the nonterminals which are reachable from the start symbol. Note that, according to items 3 and 4, $\forall p \in P_s, \text{lhs}(p) \in N_s, \text{rhs}(p) \in (N_s \cup T_s)^*$, i.e. P_s really satisfies the production set definition for a context-free grammar.

The definition of the mapping f means that some of the strings generated by G contain substrings which can be replaced with *Any*, then we obtain strings generated by G_s . In the absence of grammar simplification options developer has to work with grammar G , which can correspond to the baseline language grammar, as well as be a specially written more tolerant version of the baseline grammar, containing all the anti-patterns described explicitly. If *Any* symbol is supported by the grammar and the corresponding generator, anti-patterns forming a set P' can be substantially simplified. Symbol *Any* can be written instead of the parts denoted by γ_i in production's right hand side in case these parts satisfy the criterion 2c of the definition 2. Verification of this criterion is possible only when solving a direct problem: when the grammar G_s is generated based on the existing G . In a real situation, there is no grammar G and the developer has to solve the inverse problem: she manually writes a simplified grammar G_s , assuming that her knowledge of the particular island patterns and the general structure of the program is close to the ground truth — the structure of the baseline grammar G — and also considering parts denoted by *Any* satisfy the criterion. When this is not the case, unparsed or incorrectly parsed programs appear at the testing phase, this means that the grammar should be refined. This process usually takes several iterations.

Notice that despite the parser is built according to grammar G_s , a program from $L(G)$ is needed to be parsed. The modified LL(1) algorithm uses the criterion 2c to translate the program to the language $L(G_s)$.

B. Parsing algorithm modification

In Figure 2a the modified LL(1) parsing algorithm is presented. The highlighted lines distinguish it from the standard algorithm. In the given pseudo-code parsing stack is accessed through the *Stack* variable, input buffer is accessed through the lexical analyser object *Lexer* with methods *NextToken* returning the next token from the input stream and *CurrentToken* returning the last token that was read. The variable τ corresponds to an additional buffer for the current token, M denotes the parsing table. The grammar G_s is a regular LL(1) grammar where *Any* is a regular token, therefore parsing table construction algorithm remains unmodified and the construction itself is carried out in the standard way. Modification of the parsing algorithm is caused by the fact that parser do a more complicated job than checking if the program is valid with respect to G_s . While parser is generated by the simplified with respect to some G grammar G_s , the program derived by G comes as the input. As tokens

are received from the input stream, the modified parser should translate the program from $L(G)$ to $L(G_s)$, then it can check the syntactic correctness of the translated part.

When the terminal symbol on the top of the parsing stack does not match the current token in τ or when a nonterminal symbol X is on top of the stack and there is no record in the cell $M[X, \tau]$, the standard LL(1) algorithm reports an error because there is no explicit option available to continue parsing, and possibly starts an error recovery routine. For the modified algorithm, this situation is normal because, as it was said, the program does not belong to the language the parser is generated for. In case *Any* is on the top of the stack or the $M[X, \text{Any}]$ cell is not empty, the modified algorithm tries to replace with *Any* some sequence of tokens from the input stream, making the transition from the text from $L(G_s)$ to the text from $L(G)$. Replacement is based on the criterion 2c: the set of tokens forming the replaced sequence must not intersect with the set of tokens which are possible *Any* successors in accordance with the parsing stack state. The successors set is called *FIRST'*, it is build by the modified version of the standard *FIRST* algorithm. This modification is discussed in Section V-C. Obviously, $L(G) \subseteq L(G_s)$, because at the *Any* position not only valid $L(G)$ program subsequence can be replaced, but also an arbitrary sequence of tokens from the complement of a successors set. This makes parser less sensitive to possible errors in water regions.

It is possible to draw some parallels between the modification given and well-known error recovery algorithms: *Any* symbol looks similar to the *error* token denoting place in the grammar where recovered parsing can be resumed, *FIRST'* set seems like the set of synchronisation tokens. There are grounds for such an analogy. The program parsed is really erroneous in terms of G_s . Replacing tokens with *Any*, the parser looks for a place from which the program satisfies the grammar again. However, behind a skin-deep similarity, there is a fundamental difference in goals, implementation and results obtaining by the algorithms. Standard error recovery is performed when a program processed is clearly incorrect. The main goal of the recovery is to resume parsing at any cost. Some significant results of the previous analysis can be discarded, and a significant part of the input stream, possibly containing some points of interest, is discarded. In addition, recovery is not guaranteed to be successful. According to Section V-A, the goal of *Any* processing is the translation of a presumably valid $L(G)$ program to $L(G_s)$. The premise that the program under consideration is correct with respect to G in conjunction with the observance of the criterion 2c makes input tokens discarding totally predictable. One can be sure that the parts of the input stream replaced with *Any* belongs to the water and can be skipped without loss of the land. Furthermore, as it was previously noted, predictable and correct replacement with *Any* is possible in some cases even for programs that are incorrect with respect to G .

Further, speaking of the fact that *Any* successfully replaces a sequence of tokens of the input program, we will simply say in some cases that *Any* *matches* this sequence. Keep in mind,

```

Stack.Push($);
Stack.Push(S);

X := Stack.Peek();
t := Lexer.NextToken();
while (X ≠ $) do
  if (X = t) then
    if (t = Any) then
      Stack.Pop();
      t := Lexer.CurrentToken();
      while (t ∉ FIRST'(Stack) and t ≠ $) do
        t := Lexer.NextToken();
      end while;
      if (t = $ and $ ∉ FIRST'(Stack)) then
        error();
      end if;
    else
      Stack.Pop();
      t := Lexer.NextToken();
    end if;
  elif (M[X,t] = X → Y1Y2...Yk) then
    Stack.Pop();
    for (i from k to 1) do
      Stack.Push(Yi);
    end for;
  elif (t = Any) then
    error();
  else
    t := Any;
  end if;
  X := Stack.Peek();
end while;

if (t = $) then
  accept();
else
  error();
end if;

```

(a)

```

BuildFirst'():
  foreach (A ∈ N) do
    MemorizedFirst'[A] := ∅;
  end foreach;
  changed := true;
  while (changed) do
    changed := false;
    foreach (A → α ∈ P) do
      MemorizedFirst'[A] U= FIRST'(α);
      if (MemorizedFirst'[A] is changed) then
        changed := true;
      end if;
    end foreach;
  end while;

```

(b)

```

FIRST'(α = Y1Y2...Yk):
  first := ∅;
  for (i from 1 to k) do
    if (Yi ∈ T \ {Any}) then
      first U= {Yi};
      break;
    elif (Yi ∈ N) then
      first U= MemorizedFirst'[Yi] ∪ {ε};
      if (ε ∉ MemorizedFirst'[Yi]) then
        break;
      end if;
    end if;
  end for;
  if (∀ i ∈ [1..k]: ε ∈ MemorizedFirst'[Yi]
    or Yi = Any) then
    first U= {ε};
  end if;
  return first;

```

(c)

Fig. 2: Modified algorithms: (a) LL(1) parsing algorithm, (b) FIRST set memorization algorithm, (c) FIRST set building algorithm

that, as shown below, this process is more complex than the standard token matching.

C. The problem of consecutive "Any"

To get tokens denoting the end of the sequence that corresponds to Any, the first intention is to build the standard FIRST set for a parsing stack, treating the symbols on the stack as a string starting from its top. Unfortunately, there is a case when the standard FIRST algorithm is not enough. Sometimes two or more Any tokens can follow each other at the beginning of the sentences which can be derived from the stack. For a grammar

$A = \text{Any } B \ C; \quad B = a \mid ; \quad C = \text{Any } c ;$

the $\text{FIRST}(\text{Stack})$ set built when the first Any is processed equals to $\{a, \text{Any}\}$. The Any token is never returned by the lexical analyser, so, there is no chance that parser will recognize a string with no a tokens. As a result, a part of $L(G)$ remains uncovered by the parser, and the valid with respect

to G_s program Any Any c will never be recognized, cause there is no input program that can be transformed to it. For the example input bbbcc\$, Any processing starts at the first b and fails at the endmarker symbol \$.

To make the parser capable to cope with a simplified grammar that allows consequent Any symbols in some derivations, it is needed to modify the standard FIRST algorithm on the basis of the definition 2. According to it, Any denotes the place where the matched sequence from an input program may be empty. In the example above, the c terminal which is explicitly presented in the grammar can be treated as the end of the sequence to replace, if we assume that the sequence matched by the second Any is empty. Acting under this assumption, the modified algorithm should expand the FIRST set with the tokens that may follow the last of the subsequent Any symbols. This turns the standard FIRST set into the FIRST' used in Figure 2a.

In Figure 2b and Figure 2c, modified algorithms for FIRST' construction are presented. In Figure 2b, there is

TABLE I: Parsing table for the model example

| | a | b | c | Any | \$ |
|---|----------------------------------|---|-------------------|----------------------------------|----|
| A | $A \rightarrow B \text{ Any } C$ | | | $A \rightarrow B \text{ Any } C$ | |
| B | $B \rightarrow a$ | | | $B \rightarrow \text{Any } c$ | |
| C | | | $C \rightarrow c$ | $C \rightarrow \text{Any } b$ | |

an adopted version of the algorithm from [24, pp. 239–240]. It performs non-recursive construction of FIRST' sets for all the nonterminals in the grammar. The sets constructed are memorized in the `MemorizedFirst'` dictionary. The original algorithm is proven to be finite, the same proof is valid for the adopted version. FIRST' itself is presented in Figure 2c. Note that `Any` is not placed in the FIRST' set.

As shown in Section VI-A, when to match a sequence of `Any` is the only available option for processing some part of the input, FIRST' helps to find the actual input subsequence corresponding to the whole sequence of `Any` symbols. Technically, in this case input subsequence is matched by the first `Any`, the following `Any` symbols match empty sequences. This is the only possible solution for a simplified grammar, because to say for sure how to precisely establish a pairwise match between the parts of the input subsequence and consecutive `Any` symbols, we need more information about the original G grammar. A similar problem called *overlapping seas* is discussed in [22]: when one sea may follow another, it is impossible to distinguish between the *after-water* of the first sea and the *before-water* of the second, so the second water is believed to be empty.

The suggested FIRST' modification is proven to be enough to develop a working tolerant grammar for the real programming language.

VI. EXPERIMENTS

A. Model example

Consider the following grammar:

$A = B \text{ Any } C$; $B = a \mid \text{Any } c$; $C = \text{Any } b \mid c$;

The corresponding parsing table is presented in Table I. The rows correspond to the nonterminal symbols defined in the grammar, the columns correspond to the tokens that may appear in the buffer τ . Each cell contains the alternative that should be applied when the row nonterminal is on the top of the parsing stack and the column terminal is the lookahead token. The work of the modified parsing algorithm for a given input string is described in Table II. Each row corresponds to the iteration of the outer `while` cycle in Figure 2a, the last row corresponds to the action that takes place right after exiting the cycle. The numbers in the *Action* column correspond to the conditions numbered in Figure 2a, the number of the true condition for the current iteration is placed in the table cell together with a short description of the action performed.

This example illustrates some of the advantages of our `Any` implementation, that were declared earlier. In contrast to the situation discussed for the `Coco/R` parser generator and the

grammar in Figure 1a, at the 4th iteration, the first `a` token in the input is included in the sequence being matched by `Any`, because the `Any` symbol is really rivalled by `a` only at the 1st iteration where the choice between $B \rightarrow a$ and $B \rightarrow \text{Any } c$ productions has to be made. The 7th iteration reveals the situation specified in V-C: there is a derivation where two `Any` follow each other. Searching for all the tokens that may appear after `Any` in $A \rightarrow B \text{ Any } C$ in accordance to the parsing stack, the FIRST' algorithm looks beyond the `Any`, which is in the beginning of $C \rightarrow \text{Any } b$, and considers `b` as the possible successor of the sequence that should be matched by the current `Any`. As mentioned earlier, in case `Any` is immediately followed by other `Any` symbols, a sequence of input tokens of the maximum possible length is replaced with the first `Any`, and subsequent `Any` symbols correspond to zero-length subsequences of the input.

Dynamically performed computation of the set of symbols that may follow `Any` takes into account the actual outer context for the alternatives that are matched (this context is formed by the elements that are lower on the stack, than the current alternative), rather than all the possible outer contexts which can arise according to the grammar.

B. Real-world repositories analysis

To test the algorithm on real source code repositories, the island grammar for the C# programming language was developed. The generated parser was applied to the repositories of three industrial projects ranked from the smallest to the largest by the number of files with a source code: the LanD project itself (93 files), PascalABC.NET² (2725 files), and Roslyn³ (8027 files). PascalABC.NET is a programming language which combines Pascal syntax with .NET framework functionality. The corresponding project consists of compiler and IDE sources. Roslyn is a pair of open-source compilers for C# and Visual Basic. Roslyn project includes compiler sources and lots of test files capturing different complex and uncommon variants of a C# program. The number of files in the corresponding repositories relevant at the time of experiment conducting is given in brackets.

Rules from C# tolerant grammar are presented in Figure 3, the complete grammar can be found in LanD project repository⁴. Water rules are highlighted. Symbol $*$ denotes zero or more element repetitions, $+$ denotes one or more repetitions, $?$ denotes an optional element, brackets $()$ are used for grouping. Quantifiers of a special kind, $*!$ and $?!$, are used to set the non-empty alternative priority in case the ambiguity is detected at the parsing table construction stage. With their help, in particular, the dangling else problem is solved in the Pascal language grammar:

`if = 'if' Any 'then' operator
('else' operator) ?!`

²<https://github.com/pascalabcnet/pascalabcnet>

³<https://github.com/dotnet/roslyn>

⁴https://github.com/alexeyvaleySYRCoSE-2018/blob/master/LanD_Specifications/sharp.land

TABLE II: Tracing table

| | Stack | Input | X | t | Action | Remark |
|----|----------------|---------|-----|-----|--|---|
| 1 | \$ A | acaab\$ | A | b | (5) replace token | |
| 2 | \$ A | acaab\$ | A | Any | (3) apply $A \rightarrow B \text{ Any } C$ | |
| 3 | \$ C Any B | acaab\$ | B | Any | (3) apply $B \rightarrow \text{Any } c$ | |
| 4 | \$ C Any c Any | acaab\$ | Any | Any | (1) match <i>Any</i> , find boundary | $\text{FIRST}'(c \text{ Any } C) = \{c\}$ |
| 5 | \$ C Any c | caab\$ | c | c | (2) match <i>c</i> | |
| 6 | \$ C Any | aab\$ | Any | a | (5) replace token | |
| 7 | \$ C Any | aab\$ | Any | Any | (1) match <i>Any</i> , find boundary | $\text{FIRST}'(C) = \{b, c\}$ |
| 8 | \$ C | b\$ | C | b | (5) replace token | |
| 9 | \$ C | b\$ | C | Any | (3) apply $C \rightarrow \text{Any } b$ | |
| 10 | \$ b Any | b\$ | Any | Any | (1) match <i>Any</i> , find boundary | $\text{FIRST}'(b) = \{b\}$ |
| 11 | \$ b | b\$ | b | b | (2) match <i>b</i> | |
| 12 | \$ | \$ | \$ | \$ | (6) accept | |

```

namespace_content = opening_directive*! (attribute|namespace|namespace_member)*
opening_directive = ('using'|'extern') Any ';'
namespace= 'namespace' name '{' namespace_content '}'
namespace_member = name? (enum|delegate|class_struct_interface)
enum = 'enum' name Any '{' Any '}' ';'
delegate = 'delegate' name before_body? ';'
class_struct_interface = ('class'|'interface'|'struct') name Any '{' class_content_element* '}' ' ';
class_content_element = attribute | keyword_marked_entities
                        | name (keyword_marked_entities | class_member_tail)
keyword_marked_entities = enum | delegate | class_struct_interface | operator | event
operator = 'operator' Any arguments class_member_tail
event = 'event' name class_member_tail
class member tail = before_body? (block init value? | initializer | ';')
before_body = Any ':' (arguments|Any)*
initializer = init_expression | init_value
init_expression = '=>' (Any|block)* ';'
init_value = '=' (Any|block)* ';'
name = (ID|arguments|'extern') name_tail_element*
name_tail_element = ID|arguments|'extern'|'.'|'|'?|'|<' name_tail_element* '>'|'|' Any '}'|'|' ':'
attribute = '[' (Any|attribute)* ']'
block = '{' (Any|block)* '}'
arguments = '(' (Any|arguments)* ')'

```

Fig. 3: Rules of the tolerant C# grammar for LanD parser generator

In the C# grammar, the `*!` construct is used to distinguish between `extern` alias declaration and the header of a method written in unmanaged code. Though these constructs do not appear at the same nesting level in real programs, they are allowed to do so according to the lightweight grammar. This results in ambiguity that needs an additional priority indication.

As it can be seen, `Any` is widely used for denoting places which are insufficient for points of interest capturing. Such irrelevant areas are inheritance specification and type restrictions in class definitions (`before_block` nonterminal), field and property initializers (`initializer` nonterminal and nonterminals which are directly derivable from it). The largest parts that are matched by `Any` are blocks of code in method bodies (`block` nonterminal). A detailed description of these areas would make the grammar several times longer. In the corresponding anti-pattern formulated with `Any`, only a minimal structuring information should be placed: boundary tokens `{` and `}` are specified and self-nesting is explicitly allowed to ensure that boundaries will be matched pairwise.

TABLE III: Numbers of unparsed files per C# grammar refinement iteration

| | LanD | PascalABC.NET | Roslyn |
|---|------|---------------|--------|
| 0 | 8 | - | - |
| 1 | 0 | 39 | - |
| 2 | 0 | 0 | 209 |
| 3 | 0 | 0 | 31 |
| 4 | 0 | 0 | 2 |

This technique is also used for `attribute` and `arguments` entities, so it can be said that it forms a sustainable grammar writing pattern.

`Any` also appears in some patterns, such as `enum`, `class_struct_interface`, `operator`, denoting lakes among the land. Lakes can mark irrelevant places as well as places for which we are interested only in the list of matched tokens, not in the correct subtree specifying the deeper structure.

In Table III, the quantitative data describing the grammar

refinement process is provided. The first column contains the number of refinement iterations passed. In the table cells, there are numbers of files from each project which still cause parsing failure. Having started with the smallest project, the LanD itself, we included the bigger ones to the testing process as the grammar became refined enough to produce parser capable to parse all the files under consideration. For two refinement iterations, the number of errors for LanD and PascalABC.NET was reduced to zero. Surprisingly, even so we got a significant number of erroneously parsed and unparsed files for Roslyn (209 files out of 8027). Analysing them we found out that it was caused by tuple types and tuple literals. It is one of the new features added to C# 7.0. These constructs may look exactly like method arguments, causing confusion during parsing. The problem was solved by the less restrictive class member patterns description: the entire header is matched by the `name` pattern which includes the `arguments` pattern. The `arguments` pattern matches method arguments as well as tuple types. A more accurate division of the `name` into modifiers, type, entity name and arguments was moved at the automatically built syntax tree post-processing stage. Expression bodied properties became another cause of errors. They are widely used in Roslyn but are not presented in LanD and PascalABC.NET. To process them as the water, the `init_expression` anti-pattern was added and the `init_value` anti-pattern was refined.

At the last iteration of grammar testing and refinement, the number of errors is still non-zero. However, on closer inspection it was proved to be not a consequence of inaccuracies in the grammar structure. The first file⁵ is a test file for the Roslyn compiler, it contains the text of the program in Shift-JIS encoding, which is used for Japanese, moreover, the class name is written in Japanese. The latter causes a lexical analysis error. We consider the usage of national alphabets for entity naming to be a rare case, but, if necessary, the `ID` token can be adopted as needed. The second file⁶ also belongs to the testing infrastructure, it contains a meta-information in a form of invalid global code: there is a string field, declared directly inside the namespace but outside of the class. In the third file⁷ a code containing `using` directives and a class definition is placed after the namespace definition. This code is enclosed in `#if false` preprocessor directive, so it is not compiled after the preprocessing stage. Our tolerant parser works with the pure sources and ignores the directives, so it justifiably treat this program as incorrect.

The resulting C# grammar is aimed at all-encompassing parsing of all the possible valid C# code variations from three real-world software projects, at the same time it is both tolerant with respect to code in places indicated with `Any`, and

lightweight. For instance, the baseline C# parser description⁸ for the industrial compiler generator ANTLR, which uses an extended LL(*) algorithm [25], contains 1159 lines, and lexical analyser specification contains 1101 lines. The text of our tolerant LL(1) C# grammar has (including token definitions and different generator options) just 51 lines. Developing a parser for a certain project, one can made the grammar even more lightweight if some project-specific restrictions are known. In case some coding conventions are applied, land and water content become less variable. If a legacy code is parsed, one can be sure that the latest language features are not in use there, so the grammar is allowed not to contain patterns and anti-patterns for them.

At the next stage of the experiment, the syntax trees of the parsed files were used to calculate the numbers of successfully discovered LanD entities that we are interested in, solving the code markup task. As control numbers, the results of counting the same entities using syntax trees built by Roslyn were used. The entities were grouped into five categories: enums, classes, fields, properties, methods. The grouping is carried out in accordance with the hierarchy of classes representing the nodes of a syntactic tree in Roslyn. Entities which corresponds to Roslyn tree nodes of type `BaseFieldDeclarationSyntax` are marked as fields. These are fields themselves, as well as events described without access methods. Elements corresponding to nodes of types inherited from `BasePropertyDeclarationSyntax` are treated as properties. In addition to properties themselves, these are indexers and events with explicitly specified `add` and `remove` accessors. Methods correspond to `BaseMethodDeclarationSyntax` type: it is the parent type for method, constructor, destructor, and operator nodes.

In the Table IV, the quantitative results are presented. For all projects in all categories, LanD detects more entities than Roslyn. The difference is caused by the conditional compilation directive `#if`, which is actively used in the projects under consideration. For example, in PascalABC.NET the `#if DEBUG` construct is widely used to enable debug output and additional information collecting, conditional compilation is also presented in the sources of the syntax analysers, which are generated with GPPG.

Roslyn parser has an integrated preprocessor which resolves `#if` conditions and pass to the parsing stage only the appropriate parts of the code. LanD is a language-independent tool, so it does not have a built-in notion of directives. For a lightweight parser, directives are defined as single-line lexemes which are usually skipped. As a result, LanD statistics take into account all the entities regardless of whether or not they are enclosed in the `#if` directive with an undefined symbol. It should be noted that C# preprocessing is a fairly simple task. If necessary, the correct preprocessor can be easily written and applied to the text passed to the LanD-generated C# parser. However, this will lead to a loss of information about the areas excluded by the preprocessor.

⁵<https://github.com/dotnet/roslyn/blob/master/src/Compilers/Test/Resources/Core/Encoding/sjis.cs>

⁶<https://github.com/dotnet/roslyn/blob/master/src/Compilers/Test/Resources/Core/Symbols/Tests/Metadata/public-and-private.cs>

⁷<https://github.com/dotnet/roslyn/blob/master/src/Workspaces/Core/Portable/Shared/Extensions/ObjectExtensions.cs>

⁸<https://github.com/antlr/grammars-v4/tree/master/csharp>

TABLE IV: Number of entities found by Roslyn/LanD

| | Enums | Classes | Fields | Properties | Methods |
|---------------|---------|-------------|-------------|-------------|---------------|
| LanD | 13/14 | 94/95 | 390/390 | 248/253 | 431/436 |
| PascalABC.NET | 356/363 | 4611/4622 | 16720/16753 | 12326/12350 | 42248/42386 |
| Roslyn | 437/441 | 21583/21622 | 19606/19737 | 21886/21919 | 108040/108400 |

VII. CONCLUSION

In the present paper, the LL(1) parsing algorithm modification is proposed. This modification is intended for performing tolerant parsing based on the island grammars technique. The special `Any` symbol is integrated into the algorithm to add a capability to match token sequences which are not explicitly described in the grammar. With regard to island grammar development, the presence of `Any` simplifies the description of water and partially eliminates the need to describe the structure and variations of irrelevant areas. Besides, `Any` can be used for relevant code description in case this code contains lakes — areas for which we are interested only in pure token sequence, not in the structural information. Our `Any` implementation fixes the shortcomings of the closest analogues. It is more accurate and less restrictive in comparison with Coco/R and LightParse parser generators, it is also more simple than bounded seas approach, and still powerful enough to parse sources of large-scale software projects. It is experimentally proved that the lightweight parser of the C# language with built-in automatic construction of the syntax tree, which was developed by the authors of the current paper, makes it possible to successfully analyze the source codes of industrial software products and provides one hundred percent finding of points of interest. The developed generator of lightweight parsers is planned to be used in solving the sustainable code markup problems.

Tolerant grammar description and syntax tree post-processing are supposed to be simplified by integrating the *Schrödinger's token* concept [14] into lexical and syntax analysers. In particular, it can be useful for analyzing C# language where, along with reserved keywords, there are contextual keywords. Some of them (for example, words `where` and `partial`) directly affect the separation of land and water and the land structure analysis. Possible directions for further research are also a more intelligent resolution of the consecutive `Any` problem and integration of the `Any` symbol into LR(1) parsing algorithm.

REFERENCES

- [1] A. Goloveshkin, "Searching and analysing crosscutting concerns in marked up programming language grammar," *University News. North-Caucasian Region. Technical Sciences Series*, no. 3, pp. 29–34, Sep. 2017.
- [2] A. Afroozeh, J.-C. Bach, M. van den Brand, A. Johnstone, M. Manders, P.-E. Moreau, and E. Scott, "Island grammar-based parsing using GLL and Tom," in *Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, K. Czarnecki and G. Hedin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 224–243.
- [3] M. Van Den Brand, M. P. A. Sellink, and C. Verhoef, "Obtaining a COBOL grammar from legacy code for reengineering purposes," in *Proceedings of the 2Nd International Conference on Theory and Practice of Algebraic Specifications*, ser. Algebraic'97. Swindon, UK: BCS Learning & Development Ltd., 1997, pp. 6–6.
- [4] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE '01)*, ser. WCRE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 13–22.
- [5] —, "Lightweight impact analysis using island grammars," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2002, pp. 219–228.
- [6] S. L. Graham, C. B. Haley, and W. N. Joy, "Practical LR error recovery," *SIGPLAN Not.*, vol. 14, no. 8, pp. 168–175, Aug. 1979.
- [7] M. G. Burke and G. A. Fisher, "A practical method for LR and LL syntactic error diagnosis and recovery," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 2, pp. 164–197, Mar. 1987.
- [8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [9] M. de Jonge, E. Nilsson-Nyman, L. C. L. Kats, and E. Visser, "Natural and flexible error recovery for generated parsers," in *Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, M. van den Brand, D. Gašević, and J. Gray, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 204–223.
- [10] E. Nilsson-Nyman, T. Ekman, and G. Hedin, "Practical scope recovery using bridge parsing," in *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008, Revised Selected Papers*, D. Gašević, R. Lämmel, and E. Van Wyk, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 95–113.
- [11] R. Koppler, "A systematic approach to fuzzy parsing," *Software: Practice and Experience*, vol. 26, pp. 637–649, 1997.
- [12] P. Carvalho, N. Oliveira, and P. R. Henriques, "Unfuzzifying fuzzy parsing," in *3rd Symposium on Languages, Applications and Technologies*, ser. OpenAccess Series in Informatics (OASIs), M. J. V. Pereira, J. P. Leal, and A. Simões, Eds., vol. 38, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. Bragança, Portugal: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, June 2014, pp. 101–108.
- [13] S. Klusener and R. Lämmel, "Deriving tolerant grammars from a base-line grammar," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 179–188.
- [14] J. Aycock and R. N. Horspool, "Schrödingers token," *Software: Practice and Experience*, vol. 31, pp. 803–814, 2001.
- [15] J. Gruska, "Descriptive complexity of context-free languages," in *Mathematical Foundations of Computer Science: Proceedings of Symposium and Summer School, Strbské Pleso, High Tatras, Czechoslovakia, September 3-8, 1973*, 1973, pp. 71–83.
- [16] M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 1985.
- [17] E. Scott and A. Johnstone, "GLL parsing," *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 7, pp. 177–189, Sep. 2010.
- [18] M. Van Den Brand, P. Klint, and J. Vinju, "The syntax definition formalism SDF," [Online]. Available: <https://homepages.cwi.nl/~daybuild/daily-books/learning-about/sdf/sdf.pdf>
- [19] H. Mössenböck, "The compiler generator Coco/R," 2014. [Online]. Available: <http://ssw.jku.at/Coco/Doc/UserManual.pdf>
- [20] M. Malevannyy, "Legkovesnyi parsing i ego ispolzovanie dlya funktsii sredy razrabotki [lightweight parsing and its application in development environment]," *Informatizatsiya i svyaz [Informatization and communication]*, vol. 3, pp. 89–94, 2015, (in Russian).

- [21] M. Malevanny and S. Mikhalkovich, "Context-based model for concern markup of a source code," *Trudy ISP RAN [Proc. ISP RAS]*, vol. 28, pp. 63–78, 2016.
- [22] J. Kurš, M. Lungu, R. Iyadurai, and O. Nierstrasz, "Bounded seas," *Comput. Lang. Syst. Struct.*, vol. 44, no. PA, pp. 114–140, Dec. 2015.
- [23] B. Ford, "Packrat parsing: Simple, powerful, lazy, linear time," in *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '02. New York, NY, USA: ACM, 2002, pp. 36–47.
- [24] D. Grune and C. J. Jacobs, *Parsing Techniques: A Practical Guide (2Nd Edition)*. New York, USA: Springer-Verlag New York, 2008.
- [25] T. Parr, S. Harwell, and K. Fisher, "Adaptive LL(*) parsing: The power of dynamic analysis," *SIGPLAN Not.*, vol. 49, no. 10, pp. 579–598, Oct. 2014.

Heterogeneous Architectures Programming Library

Grigorii Kirgizov
Student

Software Engineering Department
Saint Petersburg State University
University Embankment, 7, 199034
Saint Petersburg, Russia
Email: gkirgizov@gmail.com

Iakov Kirilenko
Senior Lecturer

Software Engineering Department
Saint Petersburg State University
University Embankment, 7, 199034
Saint Petersburg, Russia
Email: y.kirilenko@spbu.ru

Abstract—Embedded platforms with heterogeneous architecture consist of a primary and one or more secondary processors. Development of software systems for these platforms poses substantial difficulties, requiring a distinct set of tools for each constituent of the heterogeneous system. It also makes achieving high efficiency the more difficult task.

This paper presents a C-like metaprogramming DSL and a library that provides a unified interface for programming secondary processors of heterogeneous systems with this DSL. Together they help to resolve aforementioned problems. The approach behind the library is a dynamic code generation: the DSL translates to LLVM IR and then compiles to native executable code at runtime.

It opens a possibility of dynamic code optimizations, e.g. runtime function specialization for specific parameters. Flexible library architecture allows simple extensibility to any target platform supported by LLVM. At the end of the paper system aprobaton on different platforms is presented.

Index Terms—metaprogramming, code generation, embedded DSL, heterogeneous systems, embedded systems

I. INTRODUCTION

Embedded systems have been in a widespread use a long time, and today they become even more relevant because of the rapid development and adoption of new application fields, for example, Internet-of-Things, "smart houses" and robotics.

Many of the embedded systems used in these areas have heterogeneous architectures due to nature of their tasks. Typically, they consist of one primary, more powerful processor which executes the main program and performs common control, and one or several secondary microcontrollers or processors that provide read/write access to sensors and peripheral devices or may perform some other special functions. Examples of such systems may be: Raspberry Pi (main) + Arduino with Atmel AVR (peripheral) and Odroid XU4 (main) + stm32f4 microcontroller (peripheral).

Heterogeneity of these systems causes noticeable overhead. Traditional development workflow requires use of IDEs and toolchains that are specific for each part of the system. This need to develop each part of the system in a separate project using a different set of platform-specific tools makes system development processes more complex and expensive. The amounts of resources required for support and changes also grow.

The efficiency of the system suffers too. Due to specificities of each microcontroller and their limited hardware capabilities they often have only basic firmware, which only capabilities are reading sensors, communicating results back to main processor, receiving data and control commands from it and writing the received data to special registers of peripheral devices. All core program logic is contained on the primary processor, and, as secondary processors/microcontrollers do not contain even a part of this logic, constant communication between them is unavoidable (because of the nature of control cycle: request sensor data, wait for it to arrive, compute control output, send it back to the secondary processors, repeat).

This work is based on preliminary results of [1] that showed the viability of the idea of dynamic code generation. We revise previous architectural choices, fully reimplement the library because of shortcomings of existing implementation and substantially extend it in terms of functionality and possible applications/uses.

In particular, the new DSL is completely abstracted from other parts of the library and can be used independently in other projects based on the idea of metaprogramming. Moreover, the new DSL implementation allows employing various dynamic optimizations which are not possible in heterogeneous systems using traditional programming techniques. The contribution of this work is twofold. We present:

- C++ embedded DSL for dynamic metaprogramming;
- a library that simplifies development of programs for heterogeneous systems providing unified programming interface; it also allows to achieve higher efficiency of the system and implement better organizations of work between its parts.

The library is based on the idea of a dynamic compilation of programs for peripheral processors.

We also demonstrate system's capabilities on a number of examples that show important features of the new DSL and some applications in embedded systems domain. Source code with build instructions can be found in the project repository¹.

Several possible use cases of this library can be imagined. First use case is avoiding the overhead of constant communication between processors. Of course, it's possible to

¹<https://github.com/gkirgizov/hetarch>

accomplish it without this library: move part of the program logic to peripheral processors on top of their basic firmware. But with usual tools, it incurs additional costs for development and support because with this approach there is no more single point of change in core logic of the system. There is unavoidable need to support several projects and ensure proper integration. Whereas presented library allows to avoid both communication overhead and unnecessary complexity of the development process.

The second use case is to allow dynamic specialization of heterogeneous systems for their operating environment. Some types of embedded heterogeneous systems can be deployed in a wide range of environments with various conditions. When their operation depends on these conditions, developers of programs for such systems must anticipate in the code all possible conditions. It may be implemented through constant monitoring of the environment. Another alternative is on-place configuration or tuning of each particular system. But it may not be possible due to nature of the task or too often or rapid (for manual operating) changes of the environment. Another variation of dynamic specialization scenario is a runtime configuration for specific peripheral devices (e.g. different models of sensors and actuators).

Our library can help there in the case of sufficiently slowly changing environment (relative to a number of control cycles, when the time required for dynamic recompilation will pay off). It can be better shown on the specific example of PID controller tuning. Firstly, PID controller with tuning subprogram is loaded on the peripheral microcontroller. Then, when optimal parameters are found, microcontroller program can be recompiled with these particular coefficients, thus yielding system that is maximally suited for its operating conditions. For the specific case of not changing environment this tuning and dynamic recompilation can be executed only once on deployment. This example is elaborated on in greater detail in the section Demonstration.

The paper is organized as follows. The next section discusses similar works that are based on the similar ideas. The third section describes main architectural decisions and presents the architecture of the system. The fourth section is devoted to the DSL and provides a reader with a number of examples. The following section describes other parts of the system and their functionality in greater detail. The Aprobation section describes test setups and the Demonstration section shows benefits of dynamic recompilation on a specific example and discusses scope and applicability of the library. The paper is closed with conclusion and discussion of possible directions of further work.

II. SIMILAR WORK

The difficulties which heterogeneous systems cause are not unique for the embedded software engineering. Programming of heterogeneous systems is an old problem, and there're several conceptual approaches to aforementioned difficulties.

The most known area that faces it is programming with graphical processors. In this case, heterogeneous system con-

sists of CPU and one or more GPUs. (The case of graphics programming, i.e. using shaders and graphics pipelines, is further from heterogeneous programming and is not considered here.) It is an old problem in this field: how to effectively and, not less importantly, conveniently use GPU in usual, CPU-centric programs? There are two main examples of systems that answer this question: Open Computing Language (OpenCL) [2] and CUDA framework from Nvidia [3]. Both these frameworks propose the use of C and C++ languages extended with special functions and attributes for writing device code (code to be executed on secondary processors). It can be written, depending on user's aims and requirements, either in separate files or in the main program files together with usual C/C++ host code that is intended to be executed on CPU. OpenCL uses dynamic compilation (at runtime) of device code; some device vendors provide offline compilers for their devices (for example, Intel Code Builder for OpenCL API). CUDA similarly provides both possibilities: Nvidia has an offline compiler called NVCC and a runtime compilation library NVRTC.

The motivation behind these examples and presented in this paper library is essentially the same: use of the same programming interface for all constituents of a heterogeneous system.

Another area that this work touches is the ideas of generative, multi-stage programming and runtime code generation. A good discussion of general motivations and trade-offs behind these ideas, as well as examples of some actual realizations and a number of references provides [4].

Among their examples Delite—a heterogeneous parallel framework for domain-specific languages [5], [6]—is of particular interest. Delite's focus is on the performance of parallel heterogeneous systems, e.g. mixed CPU/GPU architectures and clusters. It is built on top of Lightweight Modular Staging (LMS) [7] system, that makes use of a form of metaprogramming to construct a symbolic representation of a DSL program. LMS provides a basis for DSLs embedded in Scala. On top of this layer, Delite is structured into a compiler framework and a runtime component. The framework provides primitives for parallel operations and generates Scala, CUDA or C++ code from DSLs.

Although both we and the authors of Delite start from the same idea of multi-stage programming, our systems significantly differ in the approaches and application domains. Most importantly, we use dynamic code generation and thus employ the generative programming at runtime to achieve dynamic optimizations. The authors of Delite, on the other hand, require static compilation of DSLs—they promote the use of additional compilation stage to perform domain-specific optimizations.

III. HIGH LEVEL DESCRIPTION

Further in the text by the word host is meant primary processor, by target—one of the peripheral processors or microcontrollers, by the user—developer who uses this library.

A. Main Architectural Decisions

The following decisions have shown themselves as reasonable and grounded and thus are inherited from the previous work [1]. They are discussed here to provide better context.

Runtime changes in executable code on targets can be achieved by two approaches: dynamic compilation which happens on the host and code interpretation which happens on targets. Because modern interpreted languages generally have higher requirements and cause more overhead, the first decision is to use dynamic compilation on the more powerful host.

The second decision is to use embedded domain specific language (DSL) as a basis for dynamic code generation. An alternative of using code attributes with compiler extension (e.g. as used by OpenCL) is less viable due to several reasons. First, code defined in a such way can be manipulated at the runtime only as a string of characters. It complicates analysis and dynamic code specialization, requiring additional step of semantic analysis before that, whereas DSL approach gives semantic information 'for free'. Second, it's more demanding to maintain the compiler extension to keep it up-to-date with the needed compiler versions. And it's still necessary to use dynamic compilation tools. It seems excessive to support both the compiler extension and the dynamic compilation tools. Moreover, it would restrict library users to only one compiler, which can be especially inconvenient in the world of embedded systems.

LLVM [8] is used as a compilation backend. There is no real alternative, and its excellent design and convenience of use made this work possible.

C++ is chosen as a language of implementation by several reasons: firstly, it is a natural choice for embedded systems domain; secondly, it allows to avoid overhead of interfacing with LLVM; and, most importantly, with template metaprogramming it provides the necessary expressive power for implementation of the DSL, which itself must be very expressive and general to be applicable in a wide range of use cases. Specifically, the latest C++17 standard is used.

B. Architecture Overview

DSL allows the user to describe the code which will be executed on targets. CodeGen module provides a simplified interface to LLVM compilation and optimization facilities. CodeLoader, Execution and Connection modules let user load code on targets, communicate with them (for example, using global variables) and control the code execution. Management of the target's memory is provided by the host through MemoryManager module.

Fig. 1 shows the structure of the system.

This architecture has a benefit of simple extensibility. Each of the following parts of the library can be extended independently from others:

- DSL constructs and operations (for example, support array slicing or exponentiation at the language level);
- communication protocols;
- target runtime functionality;

- most importantly, target platforms.

For details on these points, the reader can proceed to the following sections.

IV. DSL

A. Design

The core of this library is a powerful embedded C-like DSL. It is translated to LLVM Intermediate Representation (IR) to allow code compilation for a wide range of targets supported by LLVM. This design of the DSL as translated and compiled at runtime is directly motivated by the concept of generative (or multi-stage) programming when the abstraction power of high-level languages is used to compose pieces of low-level code [4]. It makes runtime code generation and domain-specific optimization a fundamental part of the program logic.

As authors of [4] note, the usual appeal of DSLs is in increasing productivity by providing a higher level, more intuitive programming model for domain experts, who are not necessarily expert programmers ("user-facing" DSLs). The other direction, which is of interest for us in this paper, is in using DSL as a means for exposing knowledge about high level program structures to a compiler.

This DSL implementation makes heavy use of powerful template metaprogramming capabilities of C++, up to C++17 standard. The idea to leverage C++ templates to cope with challenges that poses development of DSLs aimed at generative programming goes back at least to the work of Czarnecki et al. [9].

B. Description and Examples

DSL provides all necessary language constructs with a familiar syntax:

- basic types (possibly cv-qualified):
 - arithmetic types;
 - pointers;
 - arrays of fixed length (possibly nested);
 - structs (possibly nested);
- operations:
 - arithmetic operators (with the support of pointer arithmetic);
 - logical operators;
 - bitwise operators;
 - C-like cast;
- control flow expressions:
 - sequential (comma operator expression);
 - conditional (if-else expression);
 - while loop;
- functions (with a fixed number of arguments; no recursion);
- literal values.

It is also easily extensible with other higher-level constructs (for example, Python-like array slicing) which will be translated directly to LLVM IR (i.e. will be efficient).

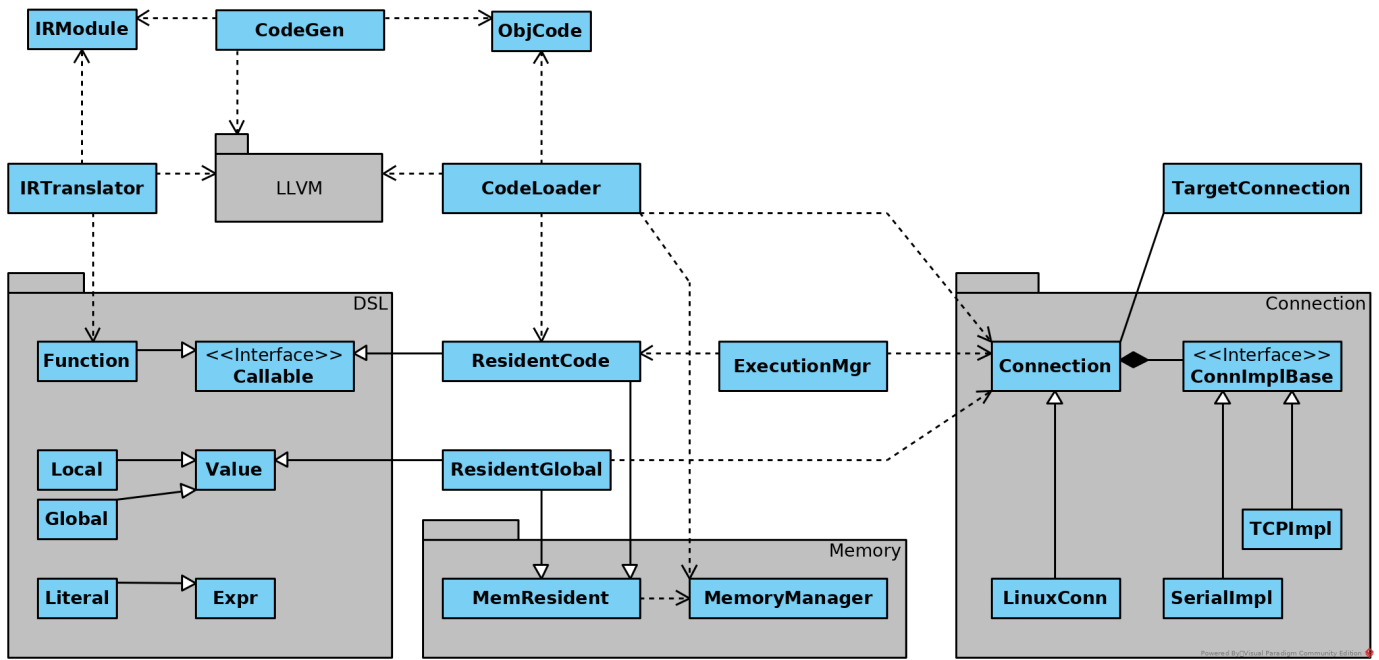


Fig. 1. UML class diagram of the system. DSL class hierarchy is shown only approximately because of its breadth and dynamic nature. IRTranslator together with non-resident DSL constructs constitute independent and reusable DSL subsystem.

To allow simpler organization of the language, every DSL construct models either value or expression; there are no statements. For example, to return void from a function user needs to use special DSL construct 'Unit'. Loops naturally return value from their last cycle. If loop didn't run it returns default-initialized value (generally, zero-initialized).

Any DSL construct has a corresponding underlying C++ type, which determines allowed operations on it and conversions to other types. Underlying C++ type can be accessed through member type alias `::type` which is present in every DSL type. And the DSL value type can be obtained (if there is one) from C++ type using `to_dsl<T>` type trait. In other words, there is a direct mapping between DSL types and C++ types. Type trait `to_dsl<T>` can be used as a convenient type factory.

Type of the DSL constructs (real C++ type, not the underlying C++ type) encodes how it was constructed and what child DSL constructs constitute it (for example see listing 1).

Listing 1. Type of some DSL expression

```
Var<int> x, y, z;
auto expr = (x + y) * z;

using expr_type =
    EBinOp< Instruction::FMul,
           EBinOp< Instruction::Add,
                Var<int>,
                Var<int>
           >,
           Var<int>
    >;
```

One of the most interesting features of the DSL is a separation of DSL abstract syntax tree (AST) construction from DSL function instantiation. It is achieved through the

use of C++14 generic lambdas which play a role of DSL code generators (AST builders). Example can be seen on the next listing.

```
auto max_gen = [](auto x, auto y) {
    return If(x > y, x, y);
};
auto dsl_max = make_dsl_fun<int, int>(max_gen);
```

It allows simple and effective reuse of needed DSL constructs, as in the next example.

```
auto max3_gen = [&](auto x1, auto x2, auto x3) {
    return max_gen(x1, max_gen(x2, x3));
};
auto dsl_max3 =
    make_dsl_fun<int, int, int>(max3_gen);
```

This conceptually differs from simple function call as a means of code reuse and is closer to function inlining. In this way the new DSL generator is constructed which, in its turn, can be later reused. Moreover, on the point of DSL code generation user can utilize C++ constructs to build more complex DSL expressions:

Listing 2. Use C++ code to build complex DSL expressions.

```
// note: accepts arbitrary DSL expressions
auto reduce_sum_gen = [](auto ...xs) {
    // Using C++17 fold expression
    return (... + xs);
};

auto sum3 = make_dsl_fun<
    float, unsigned, int
>(reduce_sum_gen);
```

Listing 3. Generator of DSL reduce function over arbitrary DSL expressions.

```
// note: accepts arbitrary DSL expressions
```

```

// (e.g. other generators)
auto get_reducer = [] (const auto& binary_op) {
    return [&] (auto x1, auto... xs) {
        // Using C++17 fold expression
        return (x1 = binary_op(x1, xs)), ... );
        // Redundant assignments
        // will be optimized out by LLVM
    };
};

auto max_vararg_gen = get_reducer(max_gen);
auto max3 = make_dsl_fun<
    float, float, float
>(max_vararg_gen);

```

Listing 4 shows two noticeable syntactic features of the DSL: the sequential operator that plays a role of C/C++ semicolon and DSL local variables. Generally, any DSL variable which is not an argument of DSL generator (enclosing lambda) will be considered a local one. For the more consistent syntax user can define local variables inside the generator lambdas. Also note that they can't be defined inside the DSL expressions because they follow the rules of C++ expressions. To use global variables a user is required to first load them on the target because they are translated to LLVM IR as actual memory addresses.

Listing 4. Use of comma operator and local variables.

```

Var<int> local1;
// note lambda capture (can also be [&])
auto max_gen = [=] (auto arg) {
    Var<int> local2;
    return (
        // variables can't be defined here!
        local1 += arg,
        local1 += local2,
        arg // last expression is returned
    );
};

```

The next listing demonstrates that DSL allows to construct complex expressions in familiar, close to C, syntax.

```

auto complex_expr = [] (Ptr<Var<uint32_t>> ptr) {
    Var<uint32_t> tmp;
    return tmp = *ptr &= ~(++ptr ^ Lit(1 << 8));
};

```

Generic DSL functions is another very useful feature. As can be seen from previous examples, DSL generators are not bound to specific types of parameters. Instead of explicit manual instantiation of DSL function with required types of parameters library user can instantiate generic DSL function with a help of function factory. If generic function is used with arguments of inappropriate types, compiler will catch this and compilation will fail with comprehensible error message.

Instantiated generic functions are stored in a function repository by a key which represents their type. As a type of DSL constructs encodes their AST, type of DSL functions encodes their body. Thus, the structural equivalence between functions is achieved without any overhead. Thanks to this repeated instantiation of the (structurally) same DSL functions is avoided. DSL function is deleted from the repository at the end of translation to LLVM IR. Needless to say, all this happens behind the scenes and a user isn't required to know about these details.

The following listing shows an example of the use of a generic DSL function.

Listing 5. Generic DSL function example

```

auto generic_max = make_generic_dsl_fun(max_gen);

auto max4_gen =
    [&] (auto x1, auto x2, auto x3, auto x4) {
        return generic_max(
            generic_max(x1, x2),
            Cast<float>(generic_max(x3, x4))
        );
    };
// This will cause instantiation
// of 2 max functions:
// for ints and for floats
auto max4 = make_dsl_fun<
    float, float, int, int
>(max4_gen);

```

Last, but not the least, DSL is designed with usability in mind. C++ code with a heavy use of templates is known for its complex error message on compilation failure. In DSL all major type constraints are checked with `static_assert` standard library function which produces comprehensible compile time error messages.

V. SUBSYSTEMS DESCRIPTION

A. MemoryManager

This centralized memory management organization allows to free less powerful targets from extra tasks and avoid extra communication cycles which would be inevitable to ensure correct memory allocation if targets managed their memory themselves. Best-fit, worst-fit and first-fit memory management algorithms are implemented. Conceptually MemoryManager is part of a CodeLoader and used only for data and code loading. That is, it's important to note that target code can't dynamically allocate memory on targets.

B. CodeLoader

With the help of CodeLoader module user can load DSL global variables and compiled code on targets. CodeLoader also allows getting a handle to already loaded variables and functions. In this case, no checks or memory allocation is performed, because, in general, there is no possibility to ensure correctness of user's actions. For example, functions can be loaded on a target in a persistent memory in one program run, and on another program run any knowledge about it will be lost, whereas the user may want to access previously loaded data and functions. So, it is assumed that user knows what he is doing.

C. Connection Module: Host side

Connection module consists of two parts: command protocol for communication between host and targets and underlying connection implementation. The functionality of the former is fully built on the primitives of the latter, which must provide synchronous read and write operations.

The core command protocol includes the following commands:

- echo (for testing);
- read specified number of bytes at a specified address;
- write data to a specified address;
- call function at the specified address (without arguments and return value);
- set function at the specified address on execution by the timer;
- set function at the specified address on execution on the specific interrupt.

This abstraction from specific implementation allows easier extensibility on new connection protocols. This work implements connection through TCP and through USB (used as a virtual serial port).

D. Connection Module: Target Runtime API

Each specific target platform requires its own firmware to interface with the host. It must provide functionality for communicating with the host and answering to requests according to the command protocol.

At this point an important consideration arises: targets must provide API sufficient for a wide range of tasks. Generally, peripheral devices on microcontrollers are memory mapped, which means that runtime API consisting of memory read and write functions can be sufficient. For example, the family of STM32 microcontrollers has fixed memory map and each device has a specific predefined address in memory.

Some platforms may need an extended API. When the target has an operating system, in particular Linux, it can additionally provide an interface to some of the system calls: `open()` for using devices represented as input/output ports and `mmap()` for correct work with library runtime process address space. It is implemented in the `LinuxConnection` module. Although for this platform it is also possible to implement an interface to arbitrary system calls and libraries using `dlopen()` and `dlsym()` functionality, the library runtime API for Linux is intentionally left minimal but sufficient for tasks concerned with controlling peripheral devices.

Another important question is a debugging interface. Issuing diagnostic messages to some local to target buffer can accommodate most of the needs and at the same time is easily implementable. Target must provide interface to read the buffer and to get an address of the target local logging function. This address is used to construct the DSL wrapper for remote logging function. From this point it can be further used in the DSL code.

VI. APROBATION

The system was tested on several setups:

- Linux on x86 plays the role of both host and target machines, communication is through TCP connection (setup for tests during development);
- the host is Linux x86, the target is Odroid XU4 (armv7a) with Linux, TCP connection;
- the host is Linux x86, the target is bare-bones stm32f429i-discovery microcontroller (armv7em), USB Virtual COM Port connection;

- the host is Odroid XU4 (armv7a) with Linux, the target is bare-bones stm32f429i-discovery (armv7em), connection through USB Virtual COM Port.

Tests were performed for each command from the command protocol (see above in the section V-C).

VII. DEMONSTRATION

For a demonstration of dynamic optimization possibilities, which this library opens, the reader can refer to the following listings of PID control (list. 6) and its tuning (list. 7) for specific conditions of the deployment environment.

Listing 6. PID controller DSL code.

```
using namespace hetarch;
using namespace hetarch::dsl;

// Convenient typedefs
// for control variables
// and coefficients
typedef int32_t ctrl_t;
typedef float coef_t;

// Example of the target
typedef uint32_t addr_t; // size_t of the target
conn::SerialConnImpl<addr_t>
  connImpl{"/dev/ttyACM0"};
SimplePipeline<addr_t>
  pipeline{"armv7e_linux_eabihf", connImpl};

// Global variables
// to store error data between control cycles
auto perr =
  pipeline.load(Global{ Var<ctrl_t>{0} });
auto ierr =
  pipeline.load(Global{ Var<ctrl_t>{0} });

// dt -- control cycle durations (in seconds)
auto pid_gen = [&](auto Kp, auto Ki, auto Kd,
                  auto dt, auto setpoint){
  auto pid_ctrl = [&]{
    // Local variables
    // pv -- process variable
    // cv -- control variable
    Var<ctrl_t> pv, cv, prev_perr, derr;

    // read_pv and write_cv
    // are some dsl generators
    // that perform actual input/output
    return (
      pv = read_pv(),

      prev_perr = perr,
      perr = setpoint - pv,
      ierr += perr,
      derr = perr - prev_perr,
      cv = Kp*perr + Kd*derr/dt + Ki*ierr*dt;

      write_cv(cv)
    );
  };
  return pid_ctrl;
};
```

Listing 7. PID tuning DSL code.

```
auto tuner = [&](auto dt, auto sp){
  // For tuning, coefficients are
  // usual mutable DSL variables
  Var<coef_t> Kp{0}, Ki{0}, Kd{0};
  auto pid_ctrl = pid_gen(Kp, Ki, Kd, dt, sp);
```

```

// Specific tuning method:
// determines current operating conditions
// (e.g. reading some sensors)
// and returns tuning data
// that allows to compute optimal
// PID controller coefficients.
// E.g. for Ziegler-Nichols method it is
// Ku -- "ultimate gain" and
// Tu -- oscillation period
return (/* actual tuning code goes here */);
};

// Example parameters
Lit sp{42}; // Setpoint
int ms_delay = 100; // Control cycle duration
Lit dt{ms_delay / 1000.0};

auto tuning_code = make_dsl_fun(tuner, dt, sp);
// Translate, compile and load tuning code
auto tuning_fun = pipeline.load(tuning_code);

// Run tuning code and get tuning data
auto tuning_data = exec.call(tuning_fun, dt, sp);
// Compute coefficients using optimal tuning data
auto coefs = compute_coefs(tuning_data);
auto Kp = coefs[0];
auto Ki = coefs[1];
auto Kd = coefs[2];

// Generate optimal PID controller
auto opt_pid_gen = pid_gen(Kp, Ki, Kd, dt, sp);
auto opt_pid_code = make_dsl_fun(opt_pid_gen);
// Translate, compile and load
// optimal PID controller
auto opt_pid = pipeline.load(opt_pid_dsl);

// Finally, run PID controller on timer
pipeline.schedule(opt_pid.callAddr, ms_delay);

```

The work is organized in the following way:

- in the first phase host loads general version of the PID controller with tuning code on the target;
- in the second phase tuning code is called and data produced by it is read by host;
- in the third phase host computes coefficients based on tuning data and recompiles PID controller with them;
- finally, host loads PID controller optimized for specific coefficients.

This example shows two advantages of using the library. Firstly, tuning code is completely absent from the final program running on the target. Dynamic code generation allows compiling code for specific constant coefficients to achieve better execution times and smaller program size.

Secondly, the dynamically generated code can be more optimal due to optimizations performed by LLVM. When coefficients are integer values, or, even better, integer powers of two (or float values, that can be rounded without big errors), resulting code will be generated with fewer (or completely without) expensive floating operations.

Listing 8. PID controller C code used for LLVM IR comparison.

```

typedef int ctrl_t;
typedef float coef_t;

extern coef_t Kp, Kd, Ki;
ctrl_t perr = 0, ierr = 0;

```

```

ctrl_t pid_ctrl(float dt, ctrl_t sp, ctrl_t pv) {
    ctrl_t prev_perr = perr;
    perr = sp - pv;
    ierr += perr;
    ctrl_t derr = perr - prev_perr;

    return Kp*perr+(Kd*derr/dt)+(Ki*ierr*dt);
}

```

To emphasize possible dynamic optimizations, fig. VII presents a comparison between listings of the PID controller code for two cases:

- C code from listing 8 compiled with clang without this library;
- DSL code from listing 6 dynamically optimized with this library.

There're several things on the fig. VII to note:

- dynamically generated code has fewer memory accesses because it is compiled for specific values (note lines 2, 7, 15 where usual code loads coefficients stored as global variables);
- instead of floating-point multiplications (lines 4 and 17 on the left) integer shift (line 4, right) and integer multiplication (line 16, right) are used;
- one apparent to a programmer optimization on line 9, right is missed: substitute multiplication by 0.5 with integer division by 2 or right shift by one; and it should be², although it is possible to implement such optimizations on the DSL level.

A. Library Applicability

The library is intended for use with embedded heterogeneous systems of a small scale with low-power secondary processors and microcontrollers that run heterogeneous tasks. The case of homogeneous tasks on the more powerful systems is better accommodated with existing tools (e.g. OpenCL or Delite) that are specifically aimed at scheduling and parallelizing the computations across bigger number of secondary processors. This library isn't intended for such use cases and doesn't provide any orchestration for parallel tasks. Each secondary processor should be managed manually and separately.

Generally, the benefits and applicability of the library should be considered in each particular case. As noted in the introduction, the library is well suited for the problems when the dynamic configuration of the system is required (either for particular environment conditions or for different peripheral devices and sensors). It's also important to consider the price of dynamic recompilation: the benefits of the specialized and optimized code should amortize the compilation price.

²This compiler behavior is expected according to C11 standard (section F9.2.1), because representations of 0.5 and 2 maybe not be equivalent and the result can be different on some machines.


```

1 ; Kp * perr
2 %9 = load float, float* @Kp
3 %10 = sitofp i32 %perr to float
4 %11 = fmul float %9, %10
5
6 ; Kd * derr / dt
7 %12 = load float, float* @Kd
8 %13 = sitofp i32 %derr to float
9 %14 = fmul float %12, %13
10 %15 = fdiv float %14, %dt
11
12 %16 = fadd float %11, %15
13
14 ; Ki * ierr * dt
15 %17 = load float, float* @Ki
16 %18 = sitofp i32 %ierr to float
17 %19 = fmul float %17, %18
18 %20 = fmul float %19, %dt
19
20 %21 = fadd float %16, %20

```

```

1
2
3 ; Kp * perr
4 %12 = shl i32 %perr, 2
5 %13 = sitofp i32 %12 to float
6
7 ; Kd * derr / dt
8 %14 = sitofp i32 %derr to float
9 %15 = fmul float %14, 5.000000e-01
10 %16 = fdiv float %15, 1.000000e-01
11
12 %17 = fadd float %16, %13
13
14
15 ; Ki * ierr * dt
16 %18 = mul i32 %ierr, 6
17 %19 = sitofp i32 %18 to float
18 %20 = fmul float %19, 1.000000e-01
19
20 %21 = fadd float %20, %17

```

Fig. 2. Comparison of LLVM IR generated for expression " $K_p * perr + (K_d * derr / dt) + (K_i * ierr * dt)$ " (core part of the PID controller code; other lines are omitted here). Compiler options used: `-O2 -target x86_64-pc-linux-gnu`. LLVM IR is used instead of native assembler because it is more readable and optimizations are done on the IR.

Left: compiled with clang from C code on listing 8. LLVM IR is presented only for the last line.

Right: compiled with LLVM from DSL (see listing 6). For the sake of demonstration it is assumed that dynamically determined PID controller coefficients are $K_p=4$, $K_d=6$, $K_i=0.5$; and control cycle duration is $dt=0.1$.

VIII. CONCLUSION

This work presented a powerful DSL language aimed at metaprogramming and showed its application to the domain of heterogeneous embedded systems. Although the library misses some features (as noted in Further Work section), it constitutes a proof of concept that the idea of dynamic code generation is perspective and useful in the real-world scenarios.

IX. FURTHER WORK

The work can be continued in several directions.

The library doesn't provide facilities for loading on the targets existing compiled code, for example, libraries. To be applicable to a wider range of use cases it requires support of this functionality.

The development of the DSL is another direction. It can be extended with additional language constructs, for example, switch, goto or to support recursion. It can also be further developed to include more features of functional programming languages, e.g. functions as first-class citizens. Support for a debugging in terms of the DSL (breakpoints, tracing) can also be added.

REFERENCES

- [1] K. Melentev, R. Belkov, and I. Kirilenko, "Sistema programmirovaniya kiberneticheskikh geterogennykh arhitektur s ispolzovaniem LLVM," in *Second Conference on Software Engineering and Information Management (SEIM-2017)(short papers)*, 2017, p. 31.
- [2] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.
- [3] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *IEEE micro*, vol. 28, no. 4, 2008.
- [4] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti *et al.*, "Go meta! A case for generative programming and DSLs in performance critical systems," in *LIPICs-Leibniz International Proceedings in Informatics*, vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- [5] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 89–100.
- [6] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 134, 2014.
- [7] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs," *Communications of the ACM*, vol. 55, no. 6, pp. 121–130, 2012.
- [8] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
- [9] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha, "DSL implementation in MetaOCaml, Template Haskell, and C++," in *Domain-Specific Program Generation*, 2003.

Applying Deep Learning to C# Call Sequence Synthesis

Aleksandr Chebykin, Iakov Kirilenko

Faculty of Mathematics and Mechanics

Saint Petersburg State University

Universitetsky prospekt, 28, Peterhof, St. Petersburg, 198504, Russian Federation

Email: a.e.chebykin@gmail.com, jake.kirilenko@gmail.com

Abstract—Many common programming tasks, like connecting to a database, drawing an image, or reading from a file, are long implemented in various frameworks and are available via corresponding Application Programming Interfaces (APIs). However, to use these functions, a software engineer must first learn of their existence and then of the correct way to utilize them. Currently, the Internet seems to be the best and the most common way to gather such information.

Recently, a deep-learning-based solution was proposed in the form of DeepAPI tool. Given English description of the desired functionality, sequence of Java function calls is generated. In this paper we show the way to apply this approach to a different programming language (C# over Java) that has smaller open code base; we describe techniques used to achieve results close to the original, as well as techniques that failed to produce an impact. Finally, we release our dataset, code and trained model to facilitate further research.

I. INTRODUCTION

When writing code, software developers often utilize various libraries via APIs. Since the problems being solved in this manner are usually similar for most users, their solutions form stable patterns of API invocations.

API mining is a long-established line of research aimed at extracting these API usage trends from source code. The importance of the task lies in the fact that generally developers spend a lot of time trying to learn frameworks' APIs in order to efficiently utilize them. A field study has found that developers often struggle to map a task from problem domain to the terminology of the API [1]. In another survey 67.6% of respondents identified that learning APIs is hindered by inadequate or absent resources [2].

Usually, when facing such problems, developers turn to general web search engines. However those are not optimized for programming-related queries and thus tend to be inefficient [3].

An alternative lies in various approaches based on statistical analysis of source code. They can provide sequences of API methods that are often used together [4], mine API specifications in the form of automata [5], synthesize relevant code snippets [6].

Deep API Learning [7] is a recent deep learning-based take on the problem that reports state-of-the-art results. The authors formulate the problem of providing API patterns satisfying users' needs as a translation one. Input language, in which user describes desired functionality, is English, and the output

language is one of API sequences: API calls are words of the language, ordered sequences of these calls form sentences. For example, English sentence "generate random int" could be translated to the language of Java API as "Random.new Random.nextInt", which corresponds to the construction of the object of type *Random* and subsequent call of its *nextInt* method.

DeepAPI tool targets exclusively Java programming language and reportedly performs well. Benefits of the approach come from the usage of deep recurrent neural networks. Thanks to them, trained model can distinguish synonyms and impact of word sequence (for example, it can distinguish queries *convert string to int* and *convert int to string*).

However the authors identify several threats to validity, including possible failure when extending the approach to other programming languages.

Our main goals are to test this threat, thus appraising generality of the approach, and to consider possible improvements. We choose C# as target languages due to its general similarity to Java, aiming to make a first step towards more different — and therefore challenging — target languages.

However even in our case simple copying of DeepAPI approach leads to bad results, and constructing well-working model proves to be far from trivial. In this paper we describe our experience of extending the proposed approach to C#.

To achieve our goals we collect dataset of 2,886,309 training samples from open source projects' code and use it to first train model with the architecture of DeepAPI (attaining the result of 10.94 BLEU), and then tune parameters to achieve BLEU 26.26. After that we introduce data preprocessing, which reduces dataset size to 1,397,597, but improves its quality and increases BLEU metric to 46.99. Finally we employ transfer learning on an alternative dataset of method names and achieve the best results of 50.14 BLEU, which is fairly close to 54.42 reported by DeepAPI on Java dataset.

Additionally we ask professional developers to evaluate output of our model on several queries, which shows that on average our model, DeepAPI#, performs as well as DeepAPI.

Our main contributions are:

- reproduction of the DeepAPI experiment with a different dataset;
- modification of the approach via programming-language-independent preprocessing which leads to results, com-

parable to original, despite lack of data;

- collection of C# dataset of commented methods and publishing of it for the benefit of the future research in the area;
- employment of transfer learning techniques for additional improvement of the results. To the best of our knowledge, we are the first to investigate transfer learning in the area of API mining.

The paper is organized as follows: in section II we outline DeepAPI model architecture. Next, in section III collection of the dataset needed for model training is discussed and additional preprocessing steps are introduced. We describe our application of transfer learning to the problem in section IV. Technical details of model training are reported in section V, which is followed by section VI, where evaluation results are described. We finish the paper with section VII, where we report work done on different related problems and discuss ways in which existing research differs from ours.

II. DEEPAPI MODEL

We borrow general model structure from DeepAPI, which is itself based on recent advancements in neural machine translation. Here we will provide only an overview, for details please refer to the original paper [7] and our previous research-in-progress paper [8].

Since the goal is to generate one sequence of words based on another, the task falls in the category of Sequence-to-Sequence learning [9]. One of the best architecture for the task is an Encoder-Decoder network [10].

It consists of two recurrent neural networks (Recurrent neural network is a special class of a neural network where unit can be connected to itself, thus allowing its state to serve the role of memory). Encoder network reads input sequence, Decoder generates output one. The process goes as following.

Encoder reads input word by word, embeds each one in a high-dimensional space and sequentially updates its hidden state, which by the end of the sentence contains language-independent idea of the input sentence. This state (also known as context vector) is then passed to the Decoder, which based on it and the last generated word generates words one by one until a special end-of-sequence token is outputted.

An example of such model at work can be seen in Figure 1. In the image states of networks are rolled out in time, so for example RNN_1, RNN_2, RNN_3 is the RNN state at time steps 1, 2, 3. Note that Encoder and Decoder consist of different RNNs and work in different time windows: at first, Encoder RNN makes 3 steps in time and then Decoder RNN makes 3 steps in time.

The benefits of this model include synonym handling (words used in the similar contexts get embedded near each other), successful processing of long inputs thanks to the memorizing ability of the recurrent networks, and finally appreciation of word sequence impact.

One major downside of such a model is the need for a large amount of sentence pairs describing the same functionality in two languages ("*generate random number*", "*Random.new*

Random.Next"). Format of the API language description is reported in the section III-A3

Source of such data can be methods' documentation comments (that in C# are XML-based and contain summary section, in which brief description of the method's functionality should be supplied) and corresponding API calls made in the method body. Details of the dataset collection are described in section III

There are several improvements of the Encoder-Decoder architecture that were shown to reliably improve results.

- Using Bidirectional Encoder leads to input being processed twice: in normal order and in reverse, resulting in 2 context vectors, which are then concatenated to get final context vector [11].
- Attention mechanism [12] allows decoder to focus on different input words when generating different output ones.

In the original DeepAPI paper an additional improvement is introduced in the form of a regularization term punishing generation of the most widespread and therefore probably problem-irrelevant API calls, such as logging ones. We have not tried such regularization since its reported impact on BLEU score is minimal. We leave testing of this enhancement for future research.

III. DATASET

A. Dataset collection

To train the model, we need to gather big amount of pairs (English description of functionality, API description). One way to do it is to process Open Source projects, looking for methods with documentation comments, extracting summary sections and linearizing interesting parts of ASTs (i.e. API calls). The processing of individual methods is described in section III-A3

GitHub¹ is one of the most popular Open Source project hostings. Following DeepAPI authors, we construct our dataset from data published there.

We attempted to augment GitHub data with data from alternative sources. In our previous paper [8] we proposed using Nuget² - a repository of compiled C# packages. However we eventually found out that compared to GitHub it does not provide much data, and what samples it provides often duplicate ones collected from GitHub. So we discontinued using Nuget as data source.

There are other sites with published open source projects, for example, Codeplex³ and SourceForge⁴. Unfortunately, we found there only a small amount of C# projects, many of which gradually migrate to GitHub, or have already done so. These hosting sites also lack search APIs, that are essential for the automatic collection of our dataset. So the potentially small

¹<http://github.com>

²<https://www.nuget.org/>

³<https://archive.codeplex.com/>

⁴<https://sourceforge.net/>

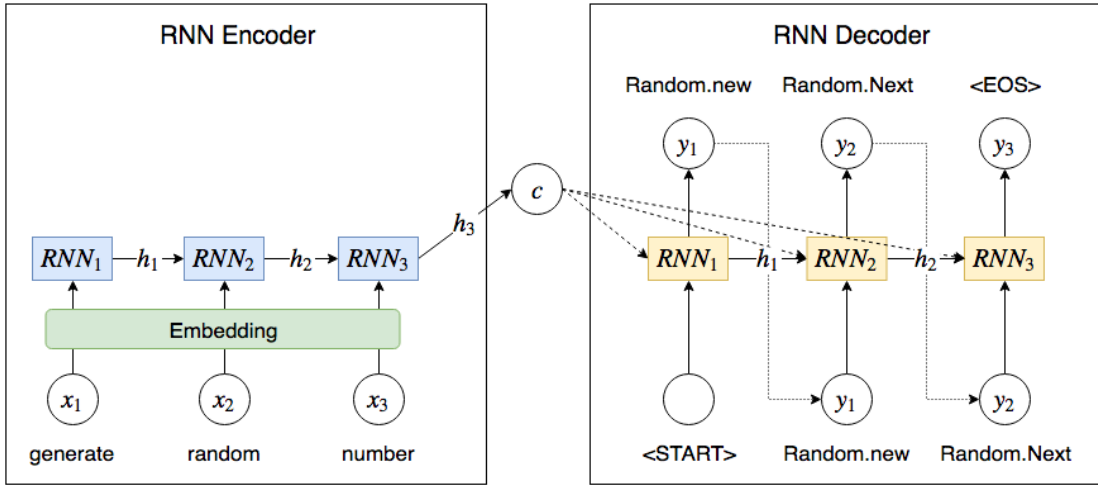


Fig. 1: RNN Encoder-Decoder workflow

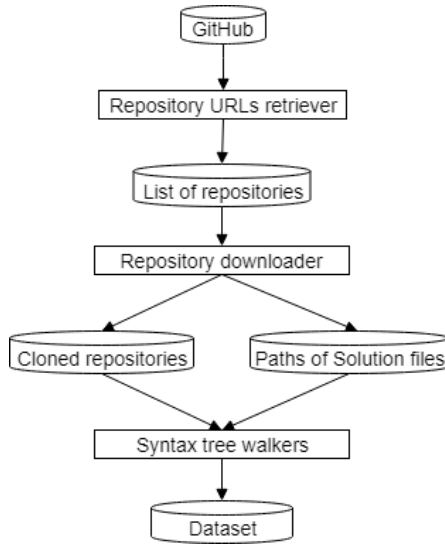


Fig. 2: Dataset gathering workflow

amount of additional data is nontrivial to collect, and therefore we choose to ignore these alternative sources.

We collect dataset from GitHub in several steps:

- 1) obtain a list of repositories relevant to us;
- 2) download these repositories;
- 3) process them, extracting from methods with documentation comments these comments, linearized in a special way API calls, types and names of method parameters.

The architecture overview can be seen in figure 2. Let us discuss every step in detail.

1) *Obtaining list of relevant repositories:* We are interested in repositories in C# language. Similar to the original paper, we would like to consider only projects that have at least one star in order to filter unused or toy projects. Both these requirements can be satisfied when setting specific parameters

of GitHub Search API.

Using this API via Octokit.rb⁵ library, we retrieve 140,990 URLs of relevant projects created from 2012 to 2017. This contrasts to the original paper that reports working with 442,928 Java repositories. So we initially have approximately 3 times less projects to work with. This lack of data can potentially be a significant obstacle when transferring the approach to other languages with smaller open code bases.

Search API also poses several technical difficulties.

Firstly, it returns no more than 1,000 results for any search request. To go around this restriction, we set additional parameter limiting repository creation date to a short span of time, for example, "2016-01-01 .. 2016-01-08". Every our requests covers 8 days, which we find short enough a period that no more than 1,000 repositories are created during it.

Secondly, Search API limits number of requests per minute by 30. In order not to exceed this limit, our script sleeps for 2 seconds after each request.

We store repositories list and the rest of our data in a SQLite database⁶.

2) *Downloading repositories:* Having gathered repository list, we can start cloning them with git. We set clone depth to 1 to speed up the process.

After download we search for solution files — special files that encompass source code files, as well as store project dependencies. We process these files in the next step.

3) *Extracting data:* C# type system is problematic for our purposes compared to Java because of the implicit type "var" introduced in version 3.0. As a consequence of its existence, code needs to be compiled in order for the type of a variable to be determined correctly, as opposed to Java where name of the variable's type or supertype is evident from its declaration.

⁵<https://github.com/octokit/octokit.rb>

⁶<https://www.sqlite.org>

| |
|--|
| <pre> /// <summary> /// Copies the data from one stream to another. /// </summary> /// <param name="from">The source stream.</param> /// <param name="to">The destination stream.</param> private void Copy(Stream from, Stream to) { var reader = new StreamReader(from); var writer = new StreamWriter(to); writer.WriteLine(reader.ReadToEnd()); writer.Flush(); } </pre> |
| <p>Comment description: copies the data from one stream to another</p> <p>API calls: System.IO.StreamReader.new System.IO.StreamWriter.new System.IO.StreamReader.ReadToEnd System.IO.StreamWriter.WriteLine System.IO.StreamWriter.Flush</p> <p>Full name: PDS.WITSLstudio.Desktop.Core.Providers.LoggingSoapExtension.Copy</p> <p>Tokenized name: logging soap extension copy</p> <p>Parameters: System.IO.Stream from, System.IO.Stream to</p> |

Fig. 3: Example of data extraction

This need for compilation limits number of projects we can process.

For compilation and syntax tree processing we use Roslyn⁷ — an open source C# compiler developed by Microsoft. To compile a project we need it to satisfy two requirements:

- 1) no manual actions are necessary for its build and compilation;
- 2) a solution file, encompassing source code files, must exist.

In order to compile more projects, we employ Nuget to restore project dependencies prior to compilation.

About 80.6% percent of repositories contain solution files, and of those 47.1% could be compiled.

After compilation we process projects in the following fashion:

- 1) find methods with documentation comments;
- 2) store whole comment and summary section;
- 3) walk syntax tree of the method body, collecting API call sequence;
- 4) store method name, parameter types and names.

An example of extracting data from method with documentation comment is provided in Figure 3

We construct API sequence similarly to the original paper. We traverse the tree in the way an interpreter might traverse it during execution, e.g. depth-first post order, processing method call's arguments before processing the call itself, and so on. When encountering constructor invocation *new C()*, we add *C.new* to the API sequence. When encountering method call *o.m()* where *o* is an instance of a class *C*, we add *C.m* to the API sequence. Additionally, when encountering *if-else* statement, we firstly process condition expression, then *if*-branch statements and finally *else*-branch statements.

We introduce one additional step to this scheme: when encountering *try-with-resources* node, we save the class *C* of an object being created in the *try* node and after processing

everything inside *try* branch we add *C.Dispose* to the API sequence. While it is easier for a programmer to rely on the language feature of *try-with-resources* block to take care of finalization of the resources, this construct is not always used, and we think that our model should know that certain sequences of API calls should end with finalization call.

Eventually we obtain 2,886,309 pairs of English descriptions and API sequences. However, this number is not directly comparable to the 7,519,907 methods reported in the DeepAPI paper. The authors explained to us (in an e-mail) that 7,519,907 is the amount of data after filtering out-of-vocabulary words, the step which in our experience removes certain samples entirely, significantly reducing size of the dataset.

Our preprocessing and the final size of dataset is discussed in the further section.

B. Data preprocessing

Upon inspecting the gathered data we conclude that it can be improved prior to being used for model training. By introducing following preprocessing steps we aim to make the training easier and the results consequently better - a notion supported by our experiments (see section VI).

1) *Language detection:* We consider our model to work with English language as input, however, many comments are not in it. Therefore we try to filter out non-English comments using language detection package⁸.

We find, however, that some English sentences are recognized as non-English. In our opinion, most likely reasons are extreme shortness of sentences used for language detection and uncommon profession-specific programmers' vocabulary. We do not want to decrease dataset size by filtering out comments incorrectly recognized as non-English, and so we change our filtering approach.

Instead of leaving only sentences recognized as English, we remove ones that are reported to be in a set of well-recognizable languages (which we deduce by hand examination) that occur in our dataset most often. Languages, sentences in which we remove, are Chinese, Korean, Japanese, Russian, German and Polish (reported in the order of decreasing frequency). As a side note, the reason for good recognition of said languages probably lies in them having alphabets different from the English one.

Such filtering leads to vocabulary containing mostly English words. It reduces training size from 2,886,309 pairs to 2,606,424.

2) *Leaving only distinct pairs:* The percent of unique pairs is about 86.6%. Note that we consider two pairs distinct even if English descriptions coincide while API descriptions do not, and vice versa.

We could identify several reasons for occurrence of repetitions:

- auto-generated code and comments (Windows Forms are especially ubiquitous);

⁷<https://github.com/dotnet/roslyn>

⁸<https://github.com/Mimino666/langdetect>

- libraries being copied to the project sources instead of being linked as dependencies.

This step reduces amount of training instances from 2,606,424 to 2,259,653.

3) *Repetition contraction*: In some API sequences an API call is repeated several times in a row. This could happen as a result of our AST linearization in a situation where, for example, an API call is made with different parameters in branches of an *if-else* statement. Since we do not record call parameters and when linearizing *if-else* statement save API calls from both branches, this may lead to an API call repeating twice in the resulting sequence. End user would not care about such repetitions in the output of the model, so we remove them before training, leaving only one copy of API call in a row.

This step does not influence amount of data, but rather is intended to improve quality of the existing training samples.

4) *Vocabulary filtering*: Similar to original paper, we create vocabularies of 10,000 most popular words in each language, and filter out the rest. If after filtering no words are left in either English description or API one, we remove the pair altogether.

This step reduces training dataset size significantly, from 2,259,653 to 1,397,597.

Additionally we experiment with, but eventually discard a preprocessing step of stemming.

5) *Stemming*: Stemming is the process of reducing inflected words to their bases. We intended to use it, as is usual, to decrease vocabulary by replacing multiple word forms with the root.

In our case it fails to provide improvement and instead makes results worse. A possible explanation may lie in the fact that stemming model was trained on regular words, not ones specific for software development and therefore works badly with this unusual vocabulary.

We discuss impact of the preprocessing steps in section VI.

The final size of our dataset is 1,397,597 pairs, which is more than 5 times smaller than 7,519,907 pairs used for training in the original paper. Even if only preprocessing from the original paper is used (i.e. vocabulary filtering and nothing else), dataset size is 1,692,898 (of which 1,434,805 pairs are unique). We consider this a significant problem that very probably makes achieving comparable results harder and takes a great toll on the model performance.

For easy reproduction of our research and for conduction of new experiments in the area, we provide our dataset⁹, as well as the code used to collect¹⁰ and preprocess¹¹ it.

IV. TRANSFER LEARNING FOR API MINING

Broadly speaking, transfer learning is utilizing knowledge gained in one problem to solve another. It is often used in

⁹<https://www.kaggle.com/awesomelemon/csharp-commented-methods-github>

¹⁰<https://github.com/AwesomeLemon/api-extraction>

¹¹<https://github.com/AwesomeLemon/api-extraction-scripts>

NLP [13] and neural machine translation, especially in the contexts where data is scarce [14]. Since our situation is one of lacking data (as shown by an experiment in section VI), we decided to investigate this idea.

A. Alternative dataset

To apply transfer learning to our problem of generating API calls given English description, we need to train a model for a task that is different, yet very similar.

As already mentioned, the DeepAPI paper proposes method bodies as source of API descriptions of functionality and method comments as source of English ones. But there is another description for a method functionality beside its comment - its name. Combined with class name, it seems descriptive of the method's contents. While not forming proper natural language sentences, these names could provide crude approximations.

Examples of correspondence between comments and names of the methods are provided in Table I. It can be seen that generally tokenized names are very similar to summary section of documentation comments. However, this is not always the case. In the last two examples despite similarity between comment and name, essential information is missing from the tokenized name. In the first of these samples key word is "Matches", without it tokenized method name loses meaning. In the second one "DWORD" is separated to "d" and "word" due to the tokenizing technique. When we tokenize method name, we assume that naming guidelines are followed and therefore first letter of the method name and first letters of every word in the name are capitalized. Here this leads to wrong division of words and thus vital information disappears, making description senseless.

However in most cases method names tokenized in this way are similar to comments and thus provide relatively good description of method contents.

We start exploration of this alternative dataset by simply training a model on it with the best parameters and our preprocessing. Resulting BLEU is not very good — 28.57 (model №4 in table II; the table is discussed minutely in section VI).

We conclude that comments indeed seem to be more descriptive of method contents than method names. But can we utilize this new dataset nonetheless?

B. Applying transfer learning for model improvement

We hypothesize that alternative method names dataset contains valuable information about correspondence between English words and API calls.

In terms of transfer learning, we can consider both our source task and target task to be the same, namely to generate API call sequence given English description of it. The difference lies in the datasets. When training for the source task, we can use the alternative dataset of pairs (Tokenized method name, API call sequence). Then we can utilize gained knowledge when training the model for the target task, that

TABLE I: Comparison between method names and comments

| Full method name | Tokenized method name | Summary section of the documentation comment |
|--|---|---|
| Method name corresponds to comment well | | |
| ManagedFusion. Serialization. JsonSerializer. Serialize | json serializer serialize | Serializes to JSON |
| MathNet.Symbolics. Packages.Standard. Structures. ComplexValue.Cosine | complex value cosine | Trigonometric Cosine of an angle in radians |
| StickyDesk. Utilities.ResizeBitmap | utilities resize bitmap | Resizes a bitmap image. |
| Nini.Config. IniConfigSource. RemoveSections | ini config source remove sections | Removes all INI sections that were removed as configs |
| Method name corresponds to comment badly | | |
| Spark.Parser. CharGrammar. StringOf | char grammar string of | Matches a string of characters |
| TagLib.Asf. DescriptionRecord. ToDWord | description record to d word | Gets the DWORD value contained in the current instance. |

makes use of the original dataset of pairs (Documentation comment summary, API call sequence).

So we train a model on the alternative dataset, and then use resulting weights for initializing the model to be trained on the standard dataset, which is a technique known as pretraining.

Also we wonder if we can similarly bootstrap learning without using alternative dataset. We perform an experiment by training the model on the comments dataset and using it for initialization and training on the same dataset.

We evaluate impact of both approaches in section VI.

V. MODEL TRAINING

Per description in section II, original authors use Encoder-Decoder architecture. As implementation of RNN they choose GRU [12]. They use 1-layered model with 1,000 hidden units and 120 dimensions for word embedding. To train the model, GroundHog¹² was used.

GroundHog since then has been discontinued, instead we use popular modern framework OpenNMT [15], that is designed specifically to train neural translation models.

We start training from the architecture reported in Deep API Learning [7]. After getting bad results we go on and empirically tune parameters, eventually arriving at following values. As RNN implementation we use LSTM [16] - a more complex model than GRU, with on-par performance, which is highly dependent on the problem. In our task it performs better. We find that 1 layer makes model not complex enough to work with C#, and since it is known that adding more layers increases model's learning ability [17], we introduce additional layers to the total of 3, which impacts results positively. We

leave number of hidden units at 1,000 and word embedding at 120 dimensions.

For training, Stochastic Gradient Descent [18] is used with batch size of 32 and exponential learning rate decay. We initialize learning rate to be 1.0 and start multiplying it by 0.7 after every epoch, starting from sixth one. Every model is trained for approximately 25 epochs on the server equipped with one Nvidia GTX 1070 GPU.

For model testing we separate 12,000 random pairs of descriptions from the dataset; the rest is used for training. We publish our trained model for easy reproduction of the results¹³.

After training, when translating queries to API sequences we follow original authors in using beam search [19], a heuristic search algorithm popular in statistical translation. Instead of generating only the most probable word on every step, we generate multiple, and then keep only several most probable sequences. This approach solves problem of discarding good translation sequences because of some sub-optimal words.

VI. EVALUATION

A. Metrics

In the area of API mining there are no universally adopted metrics. For better comparison to the original paper we follow in its steps and calculate BLEU score [20] for intrinsic evaluation, FRank score [6] and Precision@N for extrinsic one.

1) *BLEU*: BLEU is a standard metric used in machine translation to evaluate how closely generated translation resembles reference one. It does not consider grammar or others high-level features, instead calculating corrected geometric mean of n-gram precision on the whole test set [20].

Since we expect the model to generate sequences of API calls similar to the ones extracted from human-written source code, n-gram approach is applicable to our situation. The theoretical foundations of the metric stand in our case, despite target language being language of API calls rather than natural language.

BLEU is reported on the scale from 0 to 100, where higher score corresponds with bigger similarity between generated and reference sequences.

2) *FRank*: FRank metric value is the position of the first relevant result in the ranked list, as decided by a human evaluator. Such a metric is justified by two facts. Firstly, good scores of it show that the model has solved exactly the problem we intended for it, i.e. the problem of translating from English to relevant API calls. It was possible for the model to learn a target function uninteresting for us, in which case human evaluators would not find in model output API calls, relevant to the input.

Secondly, it is known that humans scan through ranked results from top to bottom [21], thus making it a desired trait for a model to rank relevant output higher.

¹²<https://github.com/pascanur/GroundHog>

¹³<http://public-resources.ml-labs.aws.intellij.net.s3.amazonaws.com/deep-api-sharp/deep-api-sharp-model.t7>

In our case FRank is measured on the scale from 1 to 10 (since similar to the DeepAPI paper, our model generates 10 outputs for every query), where lower is better. Where models fail to provide relevant results, FRank is considered to be 11.

3) *Precision@N*: Precision@N measures percentage of the relevant results in the first N outputs produced by the system. Following DeepAPI, we report Precision@5 and Precision@10 (note that the term used in the DeepAPI paper is "relevancy ratio N", which does not seem to be an established term).

This metric is reported on the scale from 0 to 100, where higher is better.

B. BLEU evaluation

In table II we report results of our experiments in terms of BLEU score. We start experiments with model architecture reported in the original paper and achieve surprisingly bad results of 10.94 BLEU, which is significantly worse than 54.42 BLEU reported in the paper. Since Java and C# are fairly similar, we expected original model to work better. Possible explanation may lie in the size of our dataset, which is more than 5 times smaller.

Model with tuned parameters achieves higher BLEU score of 26.26, which is still far from the original results.

After introduction of our preprocessing steps a 94% increase in BLEU is obtained, and the resulting score of 46.99 comes fairly close to the reported performance of the DeepAPI model.

The best result is achieved by model №8, where we employ transfer learning techniques and pretrain the model on the alternative dataset of method names. Additional analysis of transfer learning application is presented in section VI-C.

Model №4 was trained on the alternative dataset of the method names (with the size of 1,967,414 pairs) and yielded not outstanding BLEU score of 28.57. So our model performs worse on the alternative dataset, which is logical, given that it is not in grammatically correct English and sometimes does not provide good descriptions of functionality, as already discussed in section IV.

To measure if number of training instances indeed impacts model result, as we hypothesized, we try to train the model on 800,000 samples as opposed to the usual 1,397,597. This is the model №5, and it achieves 36.63 BLEU, which is worse than 46.99 achieved under the same parameters, but bigger dataset size. This leads us to the conclusion that dataset size is vital for model performance.

C. Transfer Learning evaluation

We ask several questions regarding our application of transfer learning techniques:

- 1) Does it improve our results?
- 2) Can we use the model itself for pretraining, without utilizing model trained on alternative dataset?
- 3) Is transfer learning necessary for performance improvement, or are instead our two datasets so similar that they could be merged and considered one big dataset?

TABLE II: BLEU scores for various models

| № | Parameters | Dataset | Preprocessing | Transfer learning from model № | BLEU |
|--------------------|------------|--------------------|---------------|--------------------------------|--------------|
| Parameter tuning | | | | | |
| 1 | original | comments | - | - | 10.94 |
| 2 | tuned | comments | - | - | 26.26 |
| Preprocessing data | | | | | |
| 3 | tuned | comments | yes | - | 46.99 |
| Different datasets | | | | | |
| 4 | tuned | names | yes | - | 28.57 |
| 5 | tuned | comments (part) | yes | - | 36.63 |
| 6 | tuned | comments and names | yes | - | 44.31 |
| Transfer learning | | | | | |
| 7 | tuned | comments | yes | 3 | 46.18 |
| 8 | tuned | comments | yes | 4 | 50.14 |

We answer these questions with several experiments, and come up with following answers:

- 1) Transfer learning leads to the best results achieved by us (model №8 with BLEU score of 50.14).
- 2) A model with sub-optimal parameters (which we do not include in the table in order not to clutter it) is improved by approximately 2.5 BLEU when pretrained on itself. However, best model is not, as shown by performance of model №7, that achieves only 46.18 BLEU, which approximately equals the result of the model №3 used for pretraining. So bootstrapping with the dataset itself may make sense sometimes, but not always. Presumably, model №3 was trained so well that there was no room for improvement.
- 3) We try to merge comments and names in one dataset, which we use for training model №6. Resulting BLEU score of 44.31 is better than using only names (28.57 BLEU, model №4), but worse than using only comments (46.99 BLEU, model №3). Thus we conclude that datasets are fairly different and should not be used together in a straightforward way.

D. Human evaluation

DeepAPI reports FRank, Precision@5, Precision@10 on two types of queries: popular ones, that often occur in Bing search log [6], and ones designed to showcase abilities of the proposed approach, including handling of semantically similar requests, longer input handling, combination of several tasks.

We would like to address a potential problem in evaluating the model on queries from the first group. While the DeepAPI paper reports that they do not occur in the training dataset, it seems unlikely since they were chosen for the perk of being popular, i.e. widespread, and authors do not mention filtering them out.

We test the hypothesis that such popular queries occur in the dataset by searching for them in ours. In our training data most

of these popular inputs occur multiple times as exact matches. For example, "copy file" occurs 14 times, "reverse string" occurs 7 times, "execute sql statement" occurs 14 times. We conduct this search after filtering out non-unique pairs, so for these occurrences API calls do not coincide, however, they are very similar. Therefore we believe that testing on such inputs makes little sense, because it essentially means testing on the training set, which speaks only about the model's ability to memorize. And that is expected from any model, and consequently is not very interesting.

However, to show that our model is capable of that, we test on 5 of these queries (the first 5 queries in table III).

But more interesting is the inspection of the model's ability to generalize, i.e. use gained knowledge to work with novel data. The model should be able to handle combined or semantically similar requests that are not included in the training data. We evaluate our model on 4 new queries, constructed for this exact purpose, and one such query from the DeepAPI paper. Since DeepAPI paper does not report results on 4 new queries, we used online demo of their tool¹⁴ to generate corresponding sequences.

To avoid conflict of interest, we ask 5 professional developers to evaluate extrinsic metrics for our model. Since the correspondence between query and model output is viewed differently by every developer and is up to debate, we consider relevant only those answers that were marked as relevant by at least 2 developers.

In the table III we report results of extrinsic evaluation. In general our model performs approximately the same as the original, which, having established importance of data and our lack of it, we consider an achievement.

Our model produces slightly less amount of relevant outputs (as shown by Precision@N scores), but ranks these outputs slightly higher (as shown by FRank).

Good performance on the first 5 queries demonstrate that our model is capable of memorizing correct answers, and outputs to the second 5 queries show that it can manage long requests, that require performing several action, as well as semantically similar requests.

However, both models are not very stable to slight semantic variations in the input. For example, query "create socket and then send text" is understood very well by DeepAPI, while DeepAPI# produces low amount of relevant answers, and on the contrary, query "write text using socket" perplexes DeepAPI, that generates no socket-related calls in the top 10 results, while DeepAPI# generates only relevant output.

Additionally it should be noted that while model outputs are not directly comparable due to different target languages, both models should still be able to correctly answer queries we are testing them on, since these tasks are fairly common and programming-language-independent.

E. Limitations

As already discussed in previous section, our model can be inconsistent and sensitive to query wording. While DeepAPI#

is capable of understanding synonymous queries and generating similar relevant output, it does not generate exactly the same sequences.

Also, our model is data-hungry. While we do not artificially limit our vocabulary with standard C# library, as DeepAPI does with Java and JDK, we still observe that the model cannot take into account APIs with low amount of usages. It can work with extremely popular Math.NET and Json.NET, but not with many other frameworks, even though their APIs are included in the model dictionary. It remains an open problem for the further research to find ways to make model less data-hungry, or to fine-tune it for use of specific not very popular libraries.

VII. RELATED WORK

A. API usage pattern mining

This group of projects is primarily concerned with extracting common usages of the library. The first algorithm to mine API patterns was MAPO [22]. It starts with clustering API sequences, then for every cluster finds API calls that are the most frequent there and passes those to an API usage recommender, that ranks API calls according to their similarity to the code context.

UP-Miner [23] improves upon MAPO by using API call sequence n-grams as a clustering metric and an additional clustering step. A near parameter-free approach PAM [4] significantly outperforms both MAPO and UP-Miner, introducing a probability model constructed in the form of a joint probability distribution over API calls observed in code and the underlying unobserved API patterns, used by developer. Acharya *et al.* [24] extract API patterns as partial orders, and unfortunately do not compare results to those of previous approaches.

The differences of these projects from our work are twofold. Firstly, these models do not allow user to specify their exact needs (MAPO and UP-Miner take API call as input, but an API call can be utilized in more than one scenario, therefore using it as input can be ambiguous; PAM and framework of Acharya *et al.* do not ask for input). This leads to the output containing many samples irrelevant to user, while not guaranteeing to provide those he was wishing for. Secondly, to use such models one needs to know beforehand which API calls (in case of MAPO and UP-Miner) or libraries (in case of PAM and framework of Acharya *et al.*) he is interested in. Our approach allows for recommendation of APIs to use, as well as the specifics of the usage.

B. Generating source code from natural language

Code generation based on natural language input is one of the holy grails of Computer Science. It could be seen as a more promising alternative to our problem: after all, rather than generate API call sequence and leave it to software developer to write code utilizing it, it would be better to just generate code in the first place.

However, current research in the area seems to be far from this dream. It mostly focuses on Domain Specific Languages [25], [26], which are simpler than general-purpose

¹⁴<http://211.249.63.55/>

TABLE III: Extrinsic model evaluation

| Query | DeepAPI | | | DeepAPI# | | | DeepAPI# output |
|--|---------|-----|------|----------|-----|------|---|
| | FRank | P@5 | P@10 | FRank | P@5 | P@10 | |
| convert int to string | 2 | 40 | 90 | 1 | 80 | 50 | CultureInfo.InvariantCulture Int64.ToString |
| convert string to int | 1 | 100 | 100 | 3 | 40 | 50 | Int32.TryParse |
| get current time | 10 | 10 | 10 | 1 | 60 | 40 | DateTime.Now |
| get files in folder | 3 | 40 | 50 | 1 | 80 | 90 | DirectoryInfo.GetFiles FileInfo.Name List.Add |
| generate md5 hash code | 1 | 100 | 100 | 1 | 80 | 60 | MD5.Create Encoding.GetBytes MD5.ComputeHash Byte[].Length StringBuilder.Append StringBuilder.ToString |
| copy a file and save it to a destination path | 1 | 100 | 100 | 2 | 40 | 40 | File.Exists String.Equals File.Exists IO.File.Copy |
| create socket and then send text | 1 | 100 | 90 | 3 | 20 | 10 | AddressFamily.InterNetwork SocketType.Stream ProtocolType.Tcp Socket.Connect Encoding.GetBytes Socket.Send SocketShutdown.Both Socket.Shutdown Socket.Close |
| write text using socket | - | 0 | 0 | 1 | 100 | 100 | ASCIIEncoding.GetBytes Socket.Send |
| connect to database and execute statement | 1 | 80 | 50 | 6 | 0 | 30 | IDbConnection.Open IDbConnection.CreateCommand IDbCommand.CommandText IDbCommand.ExecuteScalar Convert.ToInt32 Exception.ToString Console.WriteLine IDbConnection.Close |
| download from url and save image | 3 | 20 | 20 | 1 | 60 | 50 | String.IsNullOrEmpty WebClient.DownloadFile |
| Average scores | 3.4 | 59 | 61 | 2.0 | 56 | 52 | |

programming languages and have by definition limited usage domain.

Recent developments in generating code in general-purpose languages include works by *Ling et al.* [27] and *Yin et al.* [28]. The first paper proposes a novel approach of Latent Predictor Networks that allows for better copying of relevant key words from input to output. The second paper introduces a special version of Encoder-Decoder model (employed by us as well and described in section II), where Decoder is tailored to generate syntax trees as opposed to sequences.

The main difference between these works and ours lies in the datasets. *Ling et al.* and *Yin et al.* report results on two datasets: code of Hearthstone cards and annotated Django code (*Ling et al.* also report results on the dataset of code of Magic the Gathering cards, but this dataset is semantically very similar to the Hearthstone one). The target code for the Hearthstone dataset is rather homogenous and limited to small subset of the wide variety that is the Python language, thus resembling code in DSL more, than code in general-purpose language. And while Django dataset covers various usage scenarios, it contains impractically sesquipedalian natural language descriptions of every line of code. For example the description of line "for i in range(0, len(result)): " is "for every i in range of integers from 0 to length of result, not included". The generation of code from descriptions several times longer than itself seems impractical.

Our dataset, on the other hand, contains wide variety of API usages, described by reasonably long sentences like "Serializes to JSON", which resemble real queries written by programmers in order to look up interesting APIs.

C. Deep neural machine translation and source code

Deep API Learning [7] paper itself was published in 2016, is widely cited, but little work has followed from it. The authors went on to successfully apply the neural machine translation approach to code migration between Java and C# [29], which shows that the proposed architecture can model both languages of API sequences well.

Lin et al. [30] similarly to us apply the Encoder-Decoder approach to a different target language, specifically Bash. They succeed, but it should be noted, that their research problem is a simpler one in terms of target language used, since only 17 commands were selected from Bash. Together with command flags, types of open-vocabulary constants and logical connectives (&&, ||, parentheses) total output dictionary size does not exceed 300. To contrast that, our work is concerned with the same API dictionary size as original paper, which is 10,000 and therefore requires vastly bigger dataset and more complex model.

VIII. CONCLUSION

In this paper we applied deep learning approach for recommendation of C# APIs, removing one of the threats to the validity of the paper that originally proposed this approach for Java. To achieve this goal, we collected massive dataset, introduced several data preprocessing steps, and finally employed transfer learning techniques.

Extending DeepAPI's approach turned out to be nontrivial even for a similar language. Nonetheless, its main idea of modelling API sequences with RNN Encoder-Decoder stands.

Data preprocessing steps, suggested by us, are not dependent on C# and should therefore be applicable to any

programming language, thus they should make extending the approach even to very different languages much easier.

By releasing data, code and trained model we hope to allow repeatability of the experiments and to inspire further research in the area.

ACKNOWLEDGMENT

The authors would like to thank JetBrains Research¹⁵ for providing a GPU-equipped server for fast machine learning models training, as well as for the Young Researcher stipend granted to our team. Additionally we would like to thank Kirsanov Alexander and other friendly developers from the JetBrains ReSharper team for their input in evaluating FRank and Precision@N metrics.

REFERENCES

- [1] M. P. Robillard and R. Deline, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.
- [2] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, 2009.
- [3] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006, pp. 195–202.
- [4] J. Fowkes and C. Sutton, "Parameter-free probabilistic api mining across github," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 254–265.
- [5] S. Shoham, E. Yahav, S. J. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 651–666, 2008.
- [6] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 357–367.
- [7] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.
- [8] A. Chebykin, M. Kita, and I. Kirilenko, "Deepapi#: C# call sequence synthesis from text query."
- [9] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.
- [10] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.
- [11] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
- [12] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.
- [13] P. H. Calais Guerra, A. Veloso, W. Meira Jr, and V. Almeida, "From bias to opinion: a transfer-learning approach to real-time sentiment analysis," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 150–158.
- [14] B. Zoph, D. Yuret, J. May, and K. Knight, "Transfer learning for low-resource neural machine translation," *arXiv preprint arXiv:1604.02201*, 2016.
- [15] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," *arXiv preprint arXiv:1701.02810*, 2017.
- [16] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.
- [17] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Acoustics, speech and signal processing (icassp), 2013 IEEE international conference on*. IEEE, 2013, pp. 6645–6649.
- [18] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, pp. 462–466, 1952.
- [19] P. Koehn, "Pharaoh: a beam search decoder for phrase-based statistical machine translation models," in *Conference of the Association for Machine Translation in the Americas*. Springer, 2004, pp. 115–124.
- [20] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.
- [21] L. A. Granka, T. Joachims, and G. Gay, "Eye-tracking analysis of user behavior in www search," in *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2004, pp. 478–479.
- [22] T. Xie and J. Pei, "Mapo: Mining api usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 54–57.
- [23] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 319–328.
- [24] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 472–483.
- [25] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. Roy et al., "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 345–356.
- [26] S. Gulwani and M. Marron, "Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 803–814.
- [27] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.
- [28] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.
- [29] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deepam: Migrate apis with multi-modal sequence to sequence learning," *arXiv preprint arXiv:1704.07734*, 2017.
- [30] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, "Program synthesis from natural language using recurrent neural networks," Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep., 2017.

¹⁵<https://research.jetbrains.org/>

In-Kernel Memory-Mapped I/O Device Emulation

Vitaly Cheptsov

*Ivannikov Institute for System Programming
of the Russian Academy of Sciences;
Higher School of Economics
Moscow, Russia
cheptsov@ispras.ru*

Alexey Khoroshilov

*Ivannikov Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
khoroshilov@ispras.ru*

Abstract—Device emulation is a common necessity that arises at various steps of the development cycle, hardware migration, or reverse-engineering. While implementing the algorithms behind the device may be a nontrivial task by itself, connecting the emulator to an existing environment, such as drivers intended to work with the actual hardware, may be no less complex. Devices relying on memory-mapped input/output are of a particular interest, because unlike port-mapped input/output there is much less of a chance that the target platform provides a direct interface to intercept the transmissions. A well-known approach used in various virtual machine software is to put the entire operating system under a hypervisor and build the emulator externally. This may not be desirable for reasons like hypervisor complexity, performance loss, additional requirements for the host hardware. In this paper we extend this approach to the kernel and explain how it may be possible to build the emulator by relying on the existing interfaces provided by an operating system.

Index Terms—device emulation, memory-mapped i/o, kernel modules

I. INTRODUCTION

One of the common engineering demands is device emulation. It may arise during the software development cycle, for example, in testing or driver verification, at hardware migration, when there is no easy way to rewrite the existing software. Other than that, in the world of proprietary hardware and software it is not rare that the only way to understand and document the device abilities is to reverse-engineer it, and the ability to dynamically debug or reverse-engineer the code could be the key in security analysis or adding the device support to a virtual machine.

Speaking of virtual machines, or rather hypervisors, building the entire virtual stack for a single device one needs to emulate is often an overkill due to performance reasons, although it could be partially mitigated by hardware-assisted virtualisation and software compatibility. The latter may involve working on completely unrelated parts of the driver stack and result in unnecessary costs for continuous support.

However, while the development of full platform emulators is a considerably common topic with abundance of existing papers and products like qemu, bochs, iOS simulator, etc., peripheral emulation is much less widespread. In some cases virtual machine guest tools do try to mimic certain hardware, but even that is usually implemented as a part of a full scale platform emulation. The problem with the peripherals is not

just in implementing the algorithms behind the device, which may be a nontrivial task by itself, but also connecting the emulator to an existing environment, such as other drivers above in the stack intended to work with the real hardware.

Since one of the important aspects of using any peripherals is the ability for the CPU to communicate to them, the common demand for a device emulator is to provide a way to do it. Presently there are two common low-level approaches to perform input and output operations: port-mapped I/O (PMIO) and memory-mapped I/O (MMIO). While there are other ways such as involving some dedicated hardware, they are relatively less widespread. High-level communications operating on a packet basis (like USB bus) usually go through the dedicated abstraction layer, and thus may be implemented with the standard APIs offered by the operating system without any special effort.

It is fairly easy to implement communication protocols with a hypervisor, the standard approach is to ensure that accessing certain memory exits the virtual machine context (vmexit), which is later handled by the implementation. However, as we mentioned previously, the use of a hypervisor may be impractical, and we have to look for other means of intercepting memory access. Since direct memory access is very common, yet quite problematic to intercept, in this paper we explain how one could implement a considerably portable MMIO emulator in the kernel and cover the details of emulating device communication protocols on common platforms.

II. STATE OF THE ART

We admit to not being the first to experiment with peripheral device emulation. Every single year several published papers in the field of hardware virtualisation cover this topic to a certain level. Articles published by VMware Inc. researchers [1] [2] provide an in-depth coverage of x86-compatible hardware emulation. They explain the existing obstacles and necessary actions to be taken to implement a complete virtual stack from the CPU to network adapters. In their works they pay a lot of attention to performance optimisation, hardware-assisted virtualisation and show a visible performance penalty reduction over the new CPU generations in Virtualization nanobenchmarks section of the first referenced paper.

As a result of continuous contribution from different parties and competitive product development, the general hypervisor performance has dramatically improved. While GPU emulation is out of the scope of this paper, it should be admitted that there are several works which do manage to provide a complete GPU emulation at a reasonable performance [3] [4]. These works feature an open GPUvm platform in the Xen hypervisor.

Another related direction involves security analysis or reverse-engineering. While less frequently found in academic writing, there are several products, tools, and patches for Linux intended to log execution details from the Linux kernel for later analysis. One of the most well-known toolsets is Linux Trace Toolkit, and one of the most prominent cases of applying the approach in practice is for Nouveau driver development for NVIDIA GPUs. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack by Konstantinos Menychtas, Kai Shen, and Michael L. Scott [5] provides a good coverage in detail.

III. BASIC I/O INTRODUCTION

Port-mapped I/O is usually more demanding to the CPU instruction architecture and requires a number of so-called ports the devices will be mapped to, and perhaps a dedicated instruction set to access these ports as well. Because the device memory is accessed indirectly, another name for PMIO is detached I/O.

As an example, one of the most popular architectures to implement PMIO is x86. It can be utilised by means of two dedicated instructions: `in` and `out`, which enable one to receive and send 8, 16, or 32 bits of data to a port from 0 to 65535. Since there are faster ways to perform I/O on x86 and PMIO is not recommended for use nowadays, in some literature it may be referred to as legacy I/O. This may not be the case for other architectures found in micro-controllers, but in general MMIO support is increasing.

Memory-mapped I/O involves direct mapping of the device memory to the host memory, enabling the software to access the device just like a normal chunk of noncacheable RAM with the use of the native instruction set. Since MMIO implementation is often faster than PMIO and sometimes simpler to use, it will be the one to opt for when implementing a device communication protocol. For example, on x86 various devices installed as PCI extension cards or system management controllers make a use of it.

Virtual devices are not supposed to be functionally different from real hardware. For this reason emulators have difficulties supporting I/O communication protocols. The taken approach varies depending on the demands and available resources, but usually one of the following is used:

- Custom device development
- Driver reimplementation
- Building a hypervisor

Sadly each of these has serious limitations, and most of them create obstacles for generic peripheral emulation.

| | Device | Driver | Hypervisor |
|------------------------|--------|--------|------------|
| Software independency | + | — | ± |
| Low costs | — | ± | + |
| Legal issues | + | — | — |
| Infrastructure depend. | — | + | — |
| Forward compatibility | + | — | + |
| Performance | + | ± | ± |
| Other device support | + | + | — |

Table 1. Pros and cons summary

Developing a new device by extending a microcontroller to offer a required interface or creating an entire chip mostly works for very simple devices when a single copy is going to be used for some kind of deep debugging or instrumentation. A good example could be removable BIOS chips for debugging or HDMI to VGA adapters with HDCP decoding. While this solution is very reliable for creating a test device, the results of mass-producing a customised device will likely be not worth the effort. It will be either more expensive or worse in quality. In addition it is important to have the legal part of the question in mind and avoid patent infringement. However, this method could be most reliable when it comes to stability.

Reimplementing the driver to support another communication interface for the virtual device is very useful when working with performance-critical hardware such as GPUs. For them each extra communication layer may heavily affect the performance due to high bandwidth usage, and that is why virtualisation software implements extended GPU support (like DirectX or OpenGL) in such a way. However, in our case it defeats the entire purpose of creating a virtual device. If the point is to test the driver, it will no longer stay the same. If the reason is to support a proprietary driver, one will have to reverse-engineer it and have issues every time it gets updated.

Bringing in a virtual machine with a hypervisor is a way to overdo it. While a decent virtual machine has a wide range of supported hardware, it adds a lot of downsides as well. In particular there will always be potential performance issues, even with hardware-assisted virtualisation support. More than that, compatibility issues will likely become a blocker if the rest of the environment is not generic and well-known. It is unfortunate, but even the mainstream operating systems may be unwilling to expose new interfaces for virtual machines (like most of the graphical stack on Apple macOS).

IV. INTERCEPTING THE I/O

As a result I/O interception comes out as a pragmatic way to achieve the goal. Despite not being very common, software and hardware have enough capabilities to intercept raw device communication without touching the higher level drivers themselves.

For example, for the past 8 years the recent x86 firmwares contain a dedicated UEFI System Management Mode [6] protocol to intercept PMIO. This protocol originally existed as a `EFI_SMM_IO_TRAP_DISPATCH_PROTOCOL` protocol¹, but later on was extended with an additional

¹GUID: 58DC368D-7BFA-4E77-ABBC-0E29418DF930

IO_TRAP_EX_DISPATCH_PROTOCOL protocol². Both protocols allow you to create direct handlers to intercept the port-mapped access. By design, the management mode affects the operating system code as well, so it works throughout the boot process and is fully transparent to the higher level software implementations like OS kernel or drivers. However, aside from not being very well documented, third-party code execution in the System Management Mode is generally prohibited. So even if one is to reimplement the SMI handler similar to what Intel offers with the open source platform code, it will be of no use for anyone but UEFI firmware developers.

Fortunately, most of PMIO interface code is usually well abstracted in the kernel, and when it comes to intercepting you could just replace the underlying low level function implementation within the emulator context. However, devices relying on MMIO are of a particular interest, because unlike PMIO there is a much less chance that the target platform provides a direct interface to intercept the transmissions.

For embedded devices it may well be sufficient to statically analyse the firmware, find the instructions responsible for I/O, and either dynamically or statically overwrite them to jump to prepared thunks that will handle them accordingly. This approach is common for security analysis especially when very little is known not only about the explored peripherals but the whole controller. However, since the firmware or the driver may receive updates in the future, this approach is not very effective outside of security or code coverage analysis, and the like.

One of the first ideas that comes to mind due to the nature of MMIO writes is relying on CPU debug registers. These registers (e.g. DR on Intel or BP_CTRL/BP_COM on ARM Cortex) allow you to implement hardware breakpoints or rather watchpoints, which may trap read and write access. However, these registers are very few, and their scope area is small (i.e. a 32-bit or 64-bit word). Other than that, the kernel, debuggers, or other software may use these registers for their own needs, which leads to them being simply impractical for this kind of work.

In general-purpose operating systems with defined kernel APIs there are much better ways, such as a page protection mechanism, which is used to implement watchpoints in software. While this is suitable for doing MMIO emulation, most of the known works relying on this technique either use it for tracing or just for debugging backends. The notable example is MMIO trace in Linux, which was originally developed to reverse-engineer proprietary NVIDIA drivers by tracing the register access [7]. Other than that there hardly are very few examples of how it can be utilised for device I/O emulation.

V. PROPOSED APPROACH

The idea of general purpose I/O interception is very simple: catch reads and writes, make sure that the values read are correct, and the values written are accounted for. To apply it to MMIO we could limit page protection of the target area, and

trap the faults as they happen. Due to bandwidth limitations and architecture simplicity the I/O sequences are generally serialised, even if they happen from different threads. It may not be the case for GPUs, yet GPUs likely will not need this kind of emulation due to performance reasons. Still, in general if serialised I/O is not guaranteed even within a single memory page (which is rare) one could always implement it manually by utilising the synchronisation primitives.

Therefore, the most obvious approach will be:

1. Mark the relevant page as neither writable nor readable (not present in x86 terms).
2. Catch a fault and decode the fault address and the direction (in or out).
3. Disassemble the instruction that caused the fault and obtain its operands from the frame.
4. Handle the operands for the emulation.
5. Update the destination registers or memory for the reads as necessary.
6. Return to the location after the instruction, which caused the fault.

While it indeed solves the problem and looks very straightforward, the implementation itself could be very convoluted. While the saved context is likely to contain the fault and return addresses, bringing a full-scale disassembling framework to the kernel is inflexible due to extra architecture dependencies and considerable amounts of code required for instruction emulation. Even more, it may impose additional performance penalties, which are already tough enough.

For these reasons we tried to alter the algorithm in a way that would be simpler, less platform-dependent, and similarly performant. After examining several real-world examples we consider the following model of a MMIO-based I/O protocol, which could be applied to quite a number of devices:

1. Host ensures that the target is ready for an I/O operation.
2. Host performs the I/O operation (by reading or writing at a defined address space).
3. Host ensures that the operation is complete and repeats the process.

The 1st and 3rd steps are usually implemented as a write-and-poll, a write-and-interrupt or just as a poll. Another advantage comes out from common differences in frequencies between the host and the peripheral. Since communications usually happen between the devices with different clock bases, most of the protocols are synchronous, and the host generally does not overwrite the areas it has just written to without making a read to confirm it was successful. Even more, most of the protocols are stateful, and it is uncommon to see subsequent reads from the same place expecting the value to change more than once. A write operation will most likely appear in-between.

Under these assumptions we use a simple satisfactory transaction model as an example:

1. Write operation type (read or write).
2. Read acknowledge status until status ready.
3. Handle the values.

²GUID: 5B48E913-707B-4F9D-AF2E-EE035BCE395D

- 3.1. Read the value for read operations.
- 3.2. Write the value for write operations and read acknowledge status until status ready.

A. With write-only page support

If write-only pages are supported, in a number of cases one may implement a flip-flop approach, that will switch page protection from read-only to write-only and backwards as the process goes.

To emulate the proposed transaction we could start the communication process with the page marked as read-only, which will then trap on operation type. Here we will initiate the transaction and switch the protection to write-only. After the operation is written the trap on the status read will trigger, where we will read the written operation type, update the value for read operations and set its status. Afterwards the page protection is returned to read-only and the control is transferred back to the driver. For read operations that is all of it, for write operations the driver will read the status and attempt to perform the actual write, which should trigger the trap again. From there on one could repeat the process as described for the operation type. In the end for both reads and writes page protection returns back to read-only, eliminating any platform-specific disassembling and relying on generic approach.

Expectedly one does not have any easy access to write-only pages on popular architectures such as ARM [8] or x86. Perhaps, if these architectures were originally designed at present, when the demand for better memory protection management is much higher and when features like W^X memory and execute-only memory have already become commonplace, we would have had finer memory management that would support write-only pages. However, nowadays write-only pages are not very common in both hardware and software implementations. Certain PowerPC implementations [9] or processor extensions may provide access to them, so it remains a good idea to check CPU manuals before abandoning the try. For example, Intel x86 processors starting from Nehalem technically support write-only memory via EPT (Extended Page Tables [10]), yet it can hardly be used for anything but virtualisation.

B. Without write-only page support

When write-only pages are not available, we may still be able to work out a simpler approach, and this is where memory patching comes in hand. The idea is to let the original instruction perform the I/O just as normal, but to encode a jump-back instruction right afterwards to ensure that page protection is limited again to trap the next I/O operation. Initially this approach may appear to have too many issues to be considered in practice, however, they could all be solved with enough effort, and some of them could even be turned into benefits.

The first issue to solve is the length of the faulted instruction. A number of architectures provide fixed-length instruction sets, so the next instruction address to encode our jump-back instruction could be calculated even without knowing

anything about the current instruction. For others one could write or find simple instruction fetchers, to only decode the length without operand or operation details. Such software may also go under the name of length disassemblers, and various implementations exist for popular platforms [11]. It may become a little more involved when the I/O instruction results in non-linear control flow, but in general I/O and branching instructions belong to separate classes and are not mixed together.

The second issue occurs when the device memory is mapped to userspace and the communication happens in userspace as well. In this case a direct jump to protection restoration code is not possible, and a breakpoint or similar instruction will have to be encoded to trigger the context switch, return to the kernel and pass the control to our handler.

The third and probably the most serious issue happens when I/O operations are performed through shared code. By assuming serialised I/O we consider no cases of simultaneous code execution from the same area (unless there are multiple devices). Therefore we could safely patch it. However, nothing prohibits the driver from utilising generic memory primitives like `memcpy` or `memset` to bulk-write or read the dedicated area. These primitives generally have no effect on the I/O itself, and we do not need to intercept every byte they touch. To avoid the issue one could examine the stack trace and modify the instruction at the return address. Not only this does not require disassembling but also reduces the penalty from trapping extra I/O operations, so a quick stack unwinding that can often be implemented with compiler intrinsics easily pays off.

With all the pieces put together it creates a solid approach for a large chunk of I/O protocols. In addition to these general improvements platform-specific optimisations could be applied. For example, extra page protection changes may be avoided for write operations, if the hardware may ignore interrupts caused by write protection violation (CR0 WP bit on x86). It should be noted, that one is to pay extra attention to the scheduler (e.g. disable preemption) not to let it switch the task to another core, where write protection is on.

VI. EVALUATION

To apply the proposed solution in practice we created a software-based emulator for the 2nd generation Apple SMC in a form of a kernel extension for Apple macOS. System Management Controller (SMC) is a chip commonly found in Intel-based Apple Macintosh computers or certain Google Chromebooks. This chip is responsible for computer power management, display backlight control, HDD monitoring, thermal control, hybrid sleep and hibernation support, external device current regulation (AirPort, USB, FireWire), charging the battery, trackpad controls, screen mirroring, etc. This chip is not essential for computer functioning, and could be viewed as a convenience feature for a vendor to rely on to centralise and simplify hardware management.

There are two main generations of SMC controllers in Apple computers. The 1st generation was built on a 16-bit

Renesas H8S/2117 controller and exposed port-mapped I/O interfaces to communicate with the operating system. The 2nd and subsequent generations are based on 32-bit ARMv7-A processors, and expose memory-mapped and port-mapped I/O interfaces. Both approaches are used to implement the same functionality within a single synchronous stateful protocol. Initially the communication happens via the PMIO protocol, and then a switch to MMIO protocol happens if the device supports it. The whole communication process happens within the kernel and the existing drivers for the 2nd generation hardware are closed-source. Fortunately, due to side researchers the communication protocols are mostly documented [12].

The reasons for taking this particular device into consideration was not only because it is a challenging task compared to devices with open specifications and decent documentation, but also for the importance of having better control of the hardware you use. Apple SMC has complete access to every device in the system and could monitor the bus communications. Other than that it stores temporary encryption keys for hibernation images or user action free restarts (authenticated restarts), when full disk encryption is enabled. Apple SMC drivers expose a dedicated protocol to userspace. This protocol provides a way to obtain SMC data and configure both SMC and onboard devices. Given its direct connection to the hardware, it may be possible to inflict damage on the computer by overheating or causing power surges. Moreover, previous researches discovered that it was very easy to modify SMC firmware, which is also a very serious concern [13].

The actual implementation follows the proposed approach without write-only page support with all the suggested optimisations and certain platform-specific adjustments. SMC MMIO protocol covers a 64 KB area, which we split into pages with the dedicated handlers based on the page index. Since the access to each page is serialised, no additional I/O wrapping is necessary.

In the XNU kernel, which powers all modern Apple hardware including Macs, Intel CPU exceptions are routed through a dedicated `kernel_trap` function. To let the driver communicate with the emulated device we added a SMC nub via the standard I/O Kit APIs with mapped memory regions with restricted protection and extended the `kernel_trap` function in `EXC_I386_PGFLT` handling code specifically for our memory.

A simplified version of this code is shown in Listing 1. `ioRegionStart` and `ioRegionEnd` locate the emulated I/O area starting and ending addresses, `appleSmcStart` and `appleSmcEnd` point to the AppleSMC driver address range. `instrSize` function calculates the instruction length at the return address to later write the jump-back code via `writeTrampoline` function, which not only writes the trampoline code (by disabling the WP bit and interrupts) but additionally disables CPU preemption to avoid the scheduler switch.

To transfer the control flow to the protocol emulator `updateProtection` is performing the actual protection upgrade of the emulated I/O area and invokes the read access

```

auto faultAddr = state->cr2;
if (faultAddr >= ioRegionStart &&
    faultAddr < ioRegionEnd) {
    auto retAddr = state->rip;
    if (retAddr >= appleSmcStart &&
        retAddr < appleSmcEnd) {
        // Simple case (from AppleSMC)
        retAddr += instrSize(retAddr, 1);
    } else {
        // Complex case (from e.g. memcpy)
        retAddr = unwindToSMC(state->rsp);
    }

    auto faultType = FaultTypeRead;
    if (state->err & T_PF_WRITE) {
        faultType = FaultTypeWrite;
    }
    updateProtection(faultType, faultAddr);
    saveOrgCode(retAddr, TrampolineSize);
    writeTrampoline(faultType, faultAddr);
    return;
}

```

Listing 1. Sample code

handler. It should be noted that a dedicated procedure may be needed for platforms with delayed physical mapping update. For example, with XNU it is necessary to trigger virtual memory fault twice when the page is not present. Similarly the protection restoration routine invoked from the trampoline preserves the registers and calls the write handler.

As a result it was possible to emulate all the existing SMC protocols at no issue and avoid the use of the original device.

VII. CONCLUSION

Emulating peripheral devices within the existing operating system is not a new problem. Different solutions and approaches have appeared over the years. The industrial demand for full-stack operating system virtualisation brought their performance to a completely different level, and the needs for better customisation resulted in operating system developers providing more flexible interfaces with the possibility to create virtual hardware out of the box. Programmable microcontrollers made the process of building a device clone with the necessary features a much simpler task to accomplish.

However, there are numerous cases, where in-kernel peripheral emulation is highly anticipated, such as driver development needs, testing and verification, hardware migration, security analysis, etc. As we stated, it is often not possible or extremely impractical to attempt to incorporate virtual machines due to development costs or performance penalties. While virtual machines succeed in emulating CPUs of the same architecture at almost the same speed with hardware-assisted virtualisation, the performance of other CPUs without the use of JITs, commonly used in video game console

emulators but rarely found in generic virtualisation software, is often much worse. And in terms of I/O emulation, which is the primary concern of this paper, the situation is no better.

Furthermore, all the solutions heavily depend on the target architecture. While it was possible to think of x86 as the main architecture for personal computers in the beginning of 2000-s, today the concept of personal computers has shifted away, and other major players, e.g. ARM, appeared on the market. With this in mind the classical approach to virtualising the whole operating system could face severe issues in the future.

The idea of using page protection faults to handle device I/O events without a hypervisor may be known but not widespread anywhere out of I/O tracing. In this paper we described a way to implement a complete MMIO protocol emulator in the kernel with the use of a generic approach that has few dependencies on the target architecture and relies on platform features such as MMU and paging. We showed that certain target architecture capabilities and device protocol specifics may affect the implementation, and effectively allow or disallow a broad range of optimisations. We believe that a suggested device I/O protocol model is applicable to various hardware, and give examples on how to simplify and optimise its implementation. After exploring the existing hardware we built a SMC emulator in the XNU kernel to illustrate the suggested approach.

ACKNOWLEDGEMENTS

ISP RAS and SYRCOSE staff for review and comment. Nikita Golovliov for aid in SMC emulator development. Marvin Häuser for reverse-engineering Apple SMC UEFI drivers.

REFERENCES

- [1] Jeremy Sugerman, Ganesh Venkitachalam, Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. Proceedings of the General Track: 2001 USENIX Annual Technical Conference, 2001, pp. 1-14. <http://static.usenix.org/legacy/publications/library/proceedings/usenix01/sugerman/sugerman.ps>.
- [2] Keith Adams, Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. ASPLOS XII Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, 2006, pp. 2-13. https://www.vmware.com/pdf/asplos235_adams.pdf.
- [3] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? Proceedings of USENIX ATC '14, 2014 USENIX Annual Technical Conference, 2014, pp. 109-120. <https://www.usenix.org/system/files/conference/atc14/atc14-paper-suzuki.pdf>.
- [4] Hangchen Yu, Christopher J. Rossbach. Full Virtualization for GPUs Reconsidered. WDDD, The Annual Workshop on Duplicating, Deconstructing, and Debunking, 2017.
- [5] Konstantinos Menychtas, Kai Shen, Michael L. Scott. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. Proceedings of the 2013 USENIX conference on Annual Technical Conference, 2013, pp. 291-296. <https://www.usenix.org/system/files/conference/atc13/atc13-menychtas.pdf>.
- [6] Unified EFI, Inc. Platform Initialization (PI) Specification. Version 1.6. 2017. http://www.uefi.org/sites/default/files/resources/PI_Spec_1_6.pdf.
- [7] Jeff Muizelaar, Pekka Paalanen. In-kernel memory-mapped I/O tracing <https://www.kernel.org/doc/Documentation/trace/mmio/trace.txt>
- [8] Arm Holdings. ARM1176JZ-S Technical Reference Manual. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0333h/Caceaije.html>
- [9] NXP Semiconductors. e500mc Core Reference Manual. http://cache.freescale.com/files/32bit/doc/ref_manual/E500MCRM.pdf
- [10] Intel. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. <http://www.ece.cmu.edu/~ece845/sp17/docs/vt-overview-itj06.pdf>.
- [11] BeaEngine. Length Disassembler Engine for Intel 64-bit processors. <https://github.com/BeaEngine/lde64>
- [12] CupertinoNet. EfiPkg, AppleSmcIo protocol. <https://github.com/CupertinoNet/EfiPkg>
- [13] Crowdstrike. "Spell"unking in Apple SMC Land. 2013. http://www.nosuchcon.org/talks/2013/D1_02_Alex_Ninjas_and_Harry_Potter.pdf

Asymmetric multiprocessor problems of real-time OS

Alexander Emelenko

Ivannikov Institute for System
Programming
of the Russian Academy of Sciences
Moscow, Russian Federation
emelenko@ispras.ru

Andrey Tsyvarev

Ivannikov Institute for System
Programming
of the Russian Academy of Sciences
Moscow, Russian Federation
tsyvarev@ispras.ru

Nikolay Pakulin

Ivannikov Institute for System
Programming
of the Russian Academy of Sciences
Moscow, Russian Federation
npak@ispras.ru

Abstract— Multiprocessor support provides great advantages in increasing performance of developed products. Of course, modern operating systems must support multiprocessor execution too. However, this solution entails many problems during development, especially for real-time operating system. Our institute is working on RTOS for civil airborne avionics called JetOS and in this paper, we present our work in adding AMP (Asymmetric Multiprocessing) support.

Keywords— *real-time OS, AMP, multicore, multiprocessor*

I. INTRODUCTION

Asymmetric Multiprocessing or AMP is a system behavior description where each module, which has its own CPU, is working independently and plays its own role in system behavior. For instance, only one CPU has access to I/O operations. However, it doesn't mean that these modules can't interact.

In most cases, AMP support is the first step to a fully multiprocessing system which is our next goal in developing RTOS. Let's briefly consider the main features of our system.

JetOS is a prototype operating system for civil airborne avionics. It supports PowerPC, MIPS, ARM and x86 platforms. Also, it is designed to work within Integrated Modular Avionics (IMA) architecture and implements ARINC-653 API specification, the de-facto architecture for applied (functional) software.

The primary objectives of ARINC 653 are deterministic behavior and reliable execution of the functional software. To achieve this, ARINC-653 imposes strict requirements on time and space partitioning. For instance, all memory allocations and execution schedules are pre-defined statically.

The unit of partitioning in ARINC-653 is called partition. Every partition has its own memory space and is executed in user mode. Partitions consist of one or more processes, operating concurrently, that share the same address space. Processes have data and stack areas and they resemble the well-known concept of threads.

ARINC-653 compatible scheduler executes each partition only within time frame assigned to that partition regardless of their state. Even when all processes of an application are idle or waiting for incoming event, CPU switches to another application only when the time frame elapses. However, there are some applications (for instance, visualization application) that need as much CPU time as possible. Of course, we can

give one CPU wholly to these partitions but other CPUs must have a mechanism to communicate with them.

II. AMP CONCEPT FOR JETOS

Applicable to a real-time operating system, modules can be RTOSes (such as JetOS) with different partitions, i.e. monitor or critical plane applications; or it can be different OSes such as Linux.

In JetOS we developed the following concept of AMP support:

- Each module has its own kernel image. This kernel image consists of JetOS itself and a set of applications.
- Modules work independently on separate CPU cores, but they can exchange messages via shared memory blocks.
- Each JetOS kernel knows that there are other kernels in the system. It must use only those slices of physical memory which had been statically defined for this kernel.

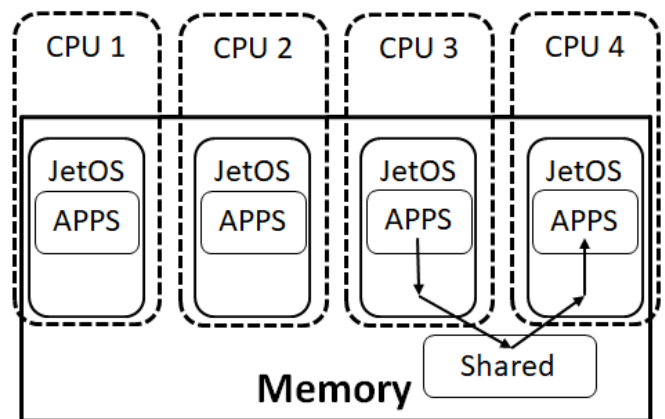


Fig. 1. JetOS AMP architecture

III. SPECIFIC CHALLENGES OF AMP FOR RTOS

Developing a multiprocessor support for a real-time OS is not a simple task. During development, we faced many challenges.

First of all, we should divide physical memory between all modules and this solution must support easy switch between configurations, i.e. adding or removing a module, changing partition number for module etc. Moreover, we must secure all

modules from each other and provide communication mechanism for them.

Secondly, fault tolerance is also important in a real-time OS. There are different situations during the flight and a critical error in one module must have no influence on other modules. Of course, we should have a mechanism to restart such modules in real time.

Thirdly, JetOS has a debugger, which uses client-server communication where the client is GDB and server is implemented in our kernel. It supports debugging both user space applications and kernel. Therefore there is a problem of debugger's behavior – whether it should stop all modules and CPUs after breakpoint in some module or not.

It is also important to mention that we have to start CPU cores with their own kernel images. Of course, we could change our kernel and use special kernel image, which would load all other kernels and continue its execution. However, our goal is to create one kernel image, which will execute different partitions without dividing kernels into the first one and the others.

Moreover, JetOS supports MIPS, x86, ARM and PowerPC architectures. All these architectures have their own multicore mechanisms, and some of them lack virtualization support. Therefore our AMP support must consider all these features. Many problems could be solved by hypervisor, but it isn't supported by some processors.

IV. DESIGN

To tackle all the above mentioned challenges and to consider all specific features, we chose paravirtualization architecture. It is a method of virtualization which requires changes in target operating systems to execute them simultaneously and independently. To achieve this, we changed JetOS in the following ways:

- During system build, we create configuration files for each module where we describe memory blocks that modules use during kernel initialization. Therefore after changing the number of modules or configuration of some module, we don't have to build all other modules, only create configuration files for them. Hence our system becomes more flexible.
- The debugger can only influence the module which is being debugged now. To debug other modules users should start another GDB client and connect to that module.
- We developed our small loader, which works both on bare-metal and in QEMU emulator. The loader is used to boot our system and it supports all architectures needed by JetOS.
- One of the goals is to distribute different devices between kernels. We use memory mapping for that purpose: each device, both PCI and embedded, has memory-mapped control and IO spaces, and a module can access a device only when corresponding memory blocks are assigned to that module.

- The biggest challenge we faced was the console: when multiple kernels want to write data on the screen, it may result in a race condition. Our solution was to give permission to communicate with the external world by serial port only to one module.
- Multiprocessor communication is realized via special shared memory.

The loader is the critical part of JetOS and it must do all the work related to CPU initialization, such as:

- Load kernel images from the file system and decompress them into physical memory.
- Start all CPUs.
- Configure necessary environment for each CPU
- Configure shared memory blocks, which are used for multi-module communication.
- Start kernels.

In some cases, it resembles a first type (or bare-metal) hypervisor, but it has many differences:

- Our loader doesn't provide any memory security. This work falls upon JetOS kernels. Kernels have strict requirements on memory management – they receive information about memory mapping and must use only specified memory.
- Also, the loader doesn't control any system calls

The JetOS AMP architecture is presented on “Fig. 1”.

V. RELATED WORK

Of course, we are not the first to consider the problem of support for asymmetric multiprocessing. There are many types of AMP solutions, such as Integrity Multivisor, XEN or JAILHOUSE, and OSes which can work with AMP such as PikeOS.

Here we briefly consider these products and their primary features.

A. Integrity Multivisor

Integrity Multivisor [1] by Green Hills software is an embedded virtualization solution. Applications and guest operating systems are flexibly scheduled across one or multiple cores. They also can communicate efficiently with each other, and utilize system peripherals.

Integrity Multivisor supports ARM, x86 and PowerPC. Also, it can work with many guest operating systems such as Linux, Windows, Solaris and VxWorks.

It provides a great capability to control execution of every OS as well as full hardware virtualization support.

The configuration is presented on “Fig. 2”.

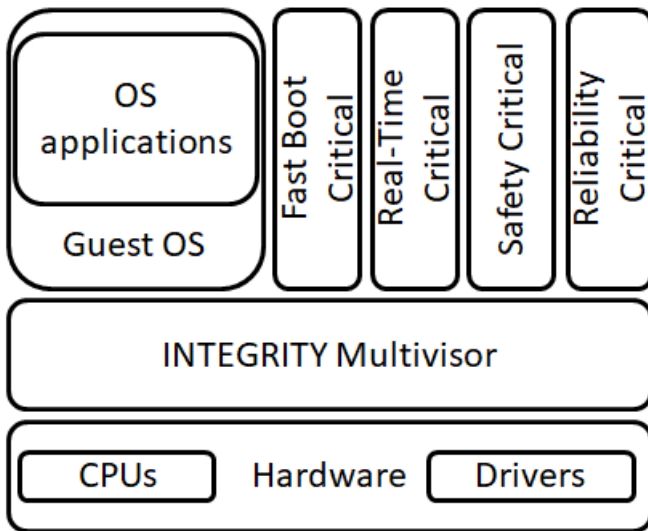


Fig. 2. Integrity Multivisor example

B. XEN

XEN project [2] is a hypervisor developed by the University of Cambridge and is now being developed by the Linux Foundation.

The main features of XEN are:

- Support for multiple guest operating systems such as Linux, Windows, NetBSD, FreeBSD and Cloud platforms – CloudStack and OpenStack.
- High level of security
- Support for both Paravirtualization (PV) and Hardware Virtualization (HVM)

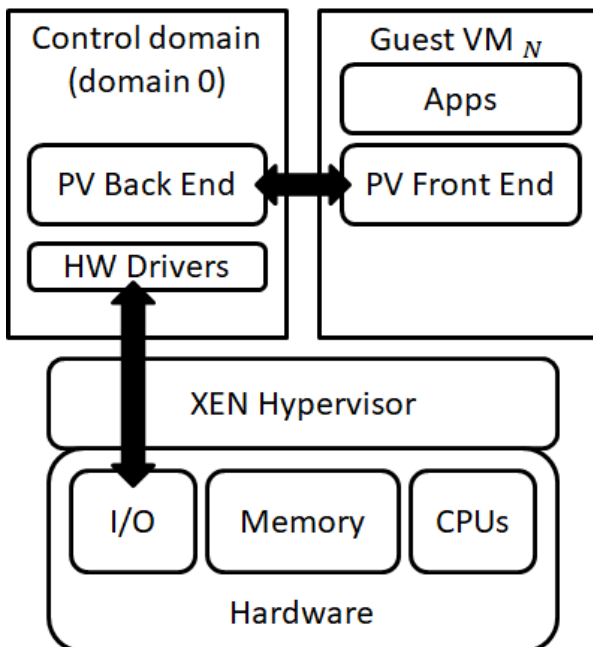


Fig. 3. XEN Paravirtualization example

XEN is available for x86, ARM, PowerPC and MIPS architectures.

It is important to say that in XEN only one module can communicate with hardware – domain 0 as on “Fig. 3”.

C. PikeOS

PikeOS by SYSGO AG [3] is a hard real-time operating system with a virtualization platform. It has been developed for security-critical applications in the fields of Aerospace, Network Infrastructures, Consumer Electronics and more.

PikeOS is designed to provide safe and secure virtualization for real-time operating systems. In AMP module, PikeOS on one CPU can coexist with various OSes on other CPU cores as on “Fig. 4”.

It guarantees independent execution of modules on different CPUs without multi-module communication.

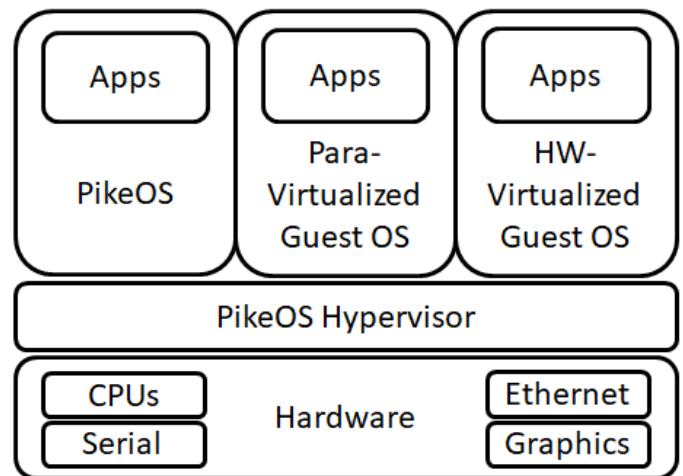


Fig. 4. PikeOS virtualization example

PikeOS design fully supports the AMP concept. Also, users can use SMP (Symmetric Multiprocessing).

PikeOS supports PowerPC, x86, ARM, MIPS, Leon 1/2 and SPARC 3/4 architectures.

D. Siemens JAILHOUSE

Siemens JAILHOUSE [4] is an open source project which presents a partitioning Hypervisor based on Linux. It's designed to work with bare-metal applications and RTOSes, as well as run real-time code. JAILHOUSE can only run on Linux.

It is also important to mention that JAILHOUSE is Asymmetric Multiprocessing and it requires that all supported architectures have at least 2 logical CPUs.

JAILHOUSE supports several platforms such as x86 and ARM.

After JAILHOUSE starts, it partitions system resources and assigns them to additional domains, called "cells". JAILHOUSE takes full control over the hardware and it

doesn't need external support. The system state can be seen on "Fig. 5".

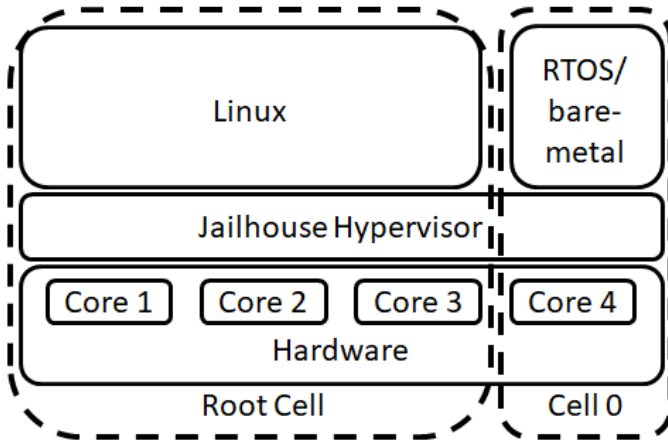


Fig. 5. Siemens Jailhouse example

JAILHOUSE doesn't support debuggers, it only provides log messages, which can be seen via a serial cable connected to real hardware or on emulator screen.

VI. FUTURE WORK

As already mentioned, AMP is a step to full multiprocessing support in JetOS.

Nowadays new avionics specifications are being developed. Some of them require SMP support. Hence, our next goal is to support SMP too.

Moreover, paravirtualization solution has a problem with memory security. So, developing a hypervisor is a good next step (of course, for those platforms, which support virtualization). It is also important to say that we can run Linux as one of the modules via hypervisor.

VII. CONCLUSION

In this paper, we presented the architecture of asymmetric multiprocessing support for JetOS hard real-time operating system. It is based on paravirtualized kernels that run on separate cores and communicate through dedicated channels in shared memory. We don't use hypervisor due to the fact that some of the platforms where JetOS runs lack hardware virtualization. In our design we support distributing devices between modules and independent static configuration of each virtual module. This architecture provides the capability to reconfigure the system in response for changes in configuration in little to no time.

Our next goals are to add SMP as well as support for SMP and AMP working simultaneously and to create hypervisor for JetOS on some of the platforms supported by JetOS.

REFERENCES

[1] Green Hills Software, "INTEGRITY Multivisor overview", 2018 https://www.ghs.com/products/rtos/integrity_virtualization.html#multivisor

[2] The Linux Foundation, "Xen Project Software Overview", 19 January 2018 https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview

[3] SYSGO AG, "SYSGO Product Overview", 2017 https://www.sysgo.com/fileadmin/user_upload/www.sysgo.com/redaktion/downloads/pdf/data-sheets/SYSGO-Product-Overview-PikeOS.pdf

[4] Siemens Corporate Technology, "Siemens JAILHOUSE". January 7, 2018 <https://github.com/siemens/jailhouse/blob/master/FAQ.md>

Building Modular Real-time software from Unified Component Model

Kurbanmagomed Mallachiev, Alexey Khoroshilov
Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn Str., Moscow, 109004, Russia.

Abstract—Modern real-time operating systems are complex embedded product made by many vendors: OS vendor, board support package vendor, device driver developers, etc. These operating systems are designed to run on different hardware; the hardware often has limited memory. Embedded OS contains many features and drivers to support different hardware. Most of the drivers are not needed for correct OS execution on a specific board. OS is statically configured to select drivers and features for each board.

Modularity of OS simplifies both configuration and development. Splitting OS to isolated modules with well specified interfaces reduces developers needs to interact during joint development. The configurator, in turn, can easily compose isolated components without component developers.

We use formal models to specify components and their composition. Formal model describes the behaviour of components and their interaction. Usage of formal models has many benefits. Models contain enough information to generate source code in C language. Our model is executable, this allows configurator to quickly verify the correctness of component configurations.

Moreover, model contains constraints on its parameters. These constraints are internal consistency or some external properties. Constraints are translated into asserts in generated source code. Therefore, we can check these constraints both at model simulation and at source code execution.

This paper presents our approach to describe such models at Scala language. We successfully tested the approach in RTOS JetOS.

I. INTRODUCTION

Modern embedded operating systems support several CPU architectures and a lot of peripheral devices. OS contains many drivers to support numerous different hardware. Embedded OS are often designed for execution in a restricted environment, for example, with limited memory. Most of the drivers are not needed for correct OS execution on some specific board and spend valuable resources. Therefore, OS must support configuration to select drivers, which will execute on the target hardware.

Static OS configuration is used in cases when it is known in advance on which hardware the OS image is going to be executed. Static means that configuration is performed on the host machine before OS loading to the target machine. The result of static OS configuration is the final image, that can be run on the target. Static configuration allows keeping final image small.

Typically, there are two roles taking part in the process of OS image building. The first role is a developer of whole OS or some driver. Developer implements his part in some

programming language, write documentation and provides support of source code and documentation. The second one is a system integrator who is responsible for correct OS configuration for specific task of specific board. Usually the system integrator does not change OS source code.

Besides simple selecting which driver will be in the final OS image many operating systems support finer tuning. For example, configuration allows selecting file system for each hard drive, or set IP address that will be used by network stack. These details are configured statically because for embedded OS and especially for safety-critical systems simplicity is more important then generality.

It is a natural desire to divide the operating system into isolated components, but not every part of the OS can be isolated. For example OS core often is strongly coupled and might be divided into isolated components only if the core will be fully redesigned to support new architecture.

If we investigate configurations of the same OS on different boards, then we will see that there is the most variable part in the OS. We call this part *OS drivers*. OS drivers contains device drivers and some services such as network stack, file system, logging, etc. Our work aimed to support flexible configuration of OS drivers.

It is common that there are many vendors involved in building of OS drivers. When services or drivers are strongly coupled, their developers have to interact a lot. Therefore, splitting OS drivers into independent isolated components helps to simplify and accelerate development.

Component should interact with each other. Appearance of fixed interface between components would make component development easier. Moreover, fixed interface can make system flexible. Only connected components can interact, and only component with the same interfaces can be connected. System integrator is responsible for connection of the components.

Suppose that system integrator created a composition of the components, which describes how each component is configured and how components are connected. We call component-based system flexible if the system integrator can:

- modify configuration of the single component without modifying others,
- substitute component with another one of the same interface without modifying other components,
- add a new component between two other connected components without modifying any component configuration except the new one.

- add to composition a copy of existing component, and they should not disturb each other.

We are developing an embedded real-time operating system for civil aircraft computers called JetOS [1]. JetOS is ARINC-653 compliant and statically configured. Approaches presented in this paper are designed for JetOS. Since JetOS is a RTOS, we are focused on minimizing the overhead added by component-based system.

II. RELATED WORKS

Classical distributed component models like Enterprise JavaBeans, CORBA and CORBA Component Model [2], [3] define components and interfaces between them. These models allow substituting one component with the another one if both have the same interfaces. Components configuration is dynamically changed by brokers. This dynamic configuration is not suitable for embedded systems with static configuration.

Ideas to separate OS appeared long ago in microkernels. Microkernel architecture's [4], [5] primary goal is to separate OS into independent servers that could be isolated from each other. Servers interact through inter-process communication (IPC). IPC calls are typed and servers with the same interface can substitute one another. But there cannot be two servers with the same interface; therefore, this model is not suitable for our tasks too.

OS-Kit [6] and eCos [7] apply modularity benefits into OS development process. They provide a set of OS components, which are used as building blocks to configure an OS. For configuration eCos uses the Component Definition Language (CDL), an extension of the existing Tool Command Language (Tcl) scripting language. Configuration is represented as feature tree with internal dependencies, group and feature constraints. Enabling of one component can lead to enable of whole components subtree. Components can have calculated value in configuration, which are calculated based on other configuration parameters. However, this is not enough for our task. Configurator cannot manage component connections and cannot add copies of the same component.

μ C/OS-II kernel uses THINK component framework [8], [9]. THINK is an implementation of the FRACTAL component model that aims to take into account the specific constraints of embedded systems development. Component describes through its interface. Interaction between components is possible once *bindings* between their interfaces has been established. Binding is a communication channel between two or more components. Binding can be created between components of a distributed system (RPC binding). This concept also does not allow to have several copies of the same component in the composition.

VxWorks is a popular embedded operating system. VxWorks board support package (BSP) is divided into components. Components interfaces are declared in Component Description Language (CDL). Note that this CDL is different from the CDL used in eCos. BSP developer can construct BSP from existing component and can add their own components.

But this system is not flexible. For example, each component has fixed list of component names it can interact with.

We are not aware of any component based model with the following set of features:

- Static configuration,
- Low overhead,
- Flexible configuration (in all aspects described in the introduction),
- Type checking of the connection, i.e. checking that connected components have the same interface.

III. COMPONENT-BASED MODEL

Our model is based on components. Component has state, which is changed during model execution, and configuration, which is immutable. Components can communicate with other components via ports. Port is a set of functions, there are two kinds of ports: *input ports* and *output ports*. Output port can be connected with input port. Set of port function signatures is called *port type*. Only input and output port of the same port type can be connected.

Each function of a component input port has an assigned handler inside the component. Call of output port function leads to the call of connected input port, which, in turn, calls the assigned handler. These calls are standard function call, or in other words synchronous call inside the same thread. Therefore, component loses control during output port call.

Thus, port call keeps the current thread. Threads cannot be created dynamically during model execution. Threads count is constant during execution.

If component needs an additional thread, then this should be explicitly specified in the model. These components are called *active*. Active components have special handlers, which are called periodically or once in the context of the new thread. We call these handlers the *activity handlers*.

In order to facilitate component reuse we introduce the concepts of a *component type* and a *component instance*. Each component type can have any number of instances. The components described above are close to component instances.

Component type contains types of component state and configuration, but not their values. Component type contains types and names of input and output ports, but not their connection. Also, component type contains implementation of:

- component initialization function, which is called at start and is used to initialise state based on the configuration;
- handlers assigned with input ports, if component has any;
- activity handlers if component is active.

Instances have unique values of state and configuration. It is easy to see that concepts of component type and component instance are similar to terms "class" and "class object" respectively.

A. Component Developer View

Component developer designs component state structure, how it should be initialized base on configuration and how it is changed during execution. Developer chooses types of configuration parameters. Developer does not aware of specific

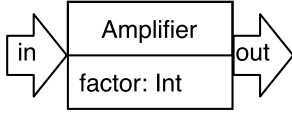


Fig. 1. Graphical representation of Amplifier component type specification

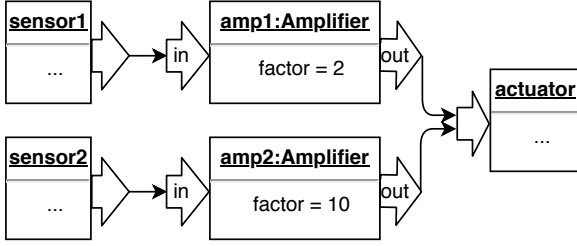


Fig. 2. Amplifier instances connection scheme.

configuration parameters values, but he can add constraints on the values. He designs component input and output ports and implements handlers for input ports. Component developer's knowledge about "outside world" is restricted by component's input and output ports. He does not know how many instances of his component will be created nor how they will be connected.

Component developer's definition of component types consists of two parts: component type specification and implementation. Specification contains:

- component type name
- component input and output port names and their types
- structure of component configuration
- description of component purpose: how it should be configured and in which environment its input ports should be called.

The rest of the information is private for component and is considered as implementation part.

B. System integrator view

System integrator gets specification of all component types in the system. System integrator decides how many instances of each component should be created and how they should be connected for solution of the specific problem. For each instance integrator sets its configuration values.

C. Simple example

Suppose that component developer created *Amplifier* component type. *Amplifier* has single input port "in" and single output port "out". Also, it has single configuration parameter "factor". Components aim is to amplify input signal from "in" port by factor "factor" and put output to "out" port.

Suppose that the system integrator wants to pass signal from two sensors to a single actuator, but he should amplify signal from first sensor by factor of 2 and from second one by factor of 10. System integrator decides to use *Amplifier* component type. He does not worry about implementation, only interfaces matters to him. For simplicity let's assume that all ports have

the same type. Amplifier component type as seen by system integrator can be seen at Fig. 1

System integrator creates two instances of *Amplifier* component type: "amp1" with configuration value "factor" equal to 2 and "amp2" with configuration value "factor" equal to 10. Then connects them accordingly to sensors and to actuator. Scheme of the result can be seen at Fig. 2

IV. PROTOTYPE

In previous work [10] we implemented component-based approach in C language with some YAML code. We used common approach to apply object-oriented ideas in C language. Component state and configuration is presented as C structure, which explicitly passed to all component functions. Calls to output ports was hidden by wrappers.

There was a lot of boilerplate code used to create component instances, describe their configuration, and their connections, in component type specification and its wrappers implementation.

To reduce amount of hand work we started to use YAML — simple declarative language. In the YAML component developer specifies component type state, configuration, input and output ports, names of functions-handler for input ports. System integrator describes in the YAML component instances, their configuration and connections. We generated C code based on these YAML specifications.

This approach has some disadvantages:

- Component developer has to manually keep consistent two files (in YAML and C languages). Change in one file leads to change in another one.
- Component developer's workflow is not comfortable: after change in YAML code generation should be processed and only then C code should be updated accordingly.
- System integrator can connect instances incorrectly (this does not apply to type checking, which is performed during compilation) and cannot see the problem until final OS image is prepared and executed in target hardware.

V. MODEL-BASED APPROACH

We decided to go further along the path of abstraction and use abstract models of components and their composition. We use formal executable models. This has many benefits. Model contains more information than source code, thus source code can be generated based on the model. Also, executable model allows to simulate instances behaviour and their interaction. This is very useful for system integrator to quickly verify the correctness of configurations. Moreover, formal model can be used to formally verify its internal consistency.

We use Scala language to model components. Scala is a functional object-oriented language that suits us well.

A. Model Description

1) *Component Developer View*: Component type is presented as Scala class inherited from interface (trait) "Component". Component configuration and state are the class fields with fixed names "config" and "state" respectively.

Active components have functions, which are called periodically or once. If component type inherits trait “RunOnce” then it should implement function “start”, which will be called once after component initialization. If component type inherits class “Periodically”, then it should implement function “periodically”; the frequency of the call is determined by the configuration.

For example, consider “Counter” component type, which has a state but no configuration. State contains value “callCount”, which is initialized with zero. Function “periodically” increases “callCount” on every call.

```
class Counter extends Periodic with Component {
  class State(val callCount: Int)
  var state = new State(0)

  type Config = Unit
  val config = ()

  def periodically = {callCount += 1}
}
```

Listing 1. “Counter” component type.

Port types are declared as interfaces(traits). Input ports are defined inside component type class as objects, which inherited port type. Output port are class fields with type of port type. Output ports values are passed as component type constructor parameters. It is worth noting that output ports can be passed by name to constructor, this allows initializing component instances with cycle connections among them.

Example of input/output ports for “Amplifier” component type (defined in previous sections):

```
trait SignalProcessor {
  def processSignal(s: Int): Int
}

class Amplifier(out: =>SignalProcessor)...{
  ...
  object in extends SignalProcessor {
    def processSignal(s: Int): Int = {
      val processed = process(s)
      out.processSignal(processed)
    }
  }
}
```

Listing 2. Port type SignalProcessor and ports of “Amplifier” component type. The component type has input port “in” and output port “out”, both of them have type SignalProcessor. There is an implementation of function processSignal of “in” port. Port “out” passed-by-name. Scala syntax may be confusing, here function processSignal returns result of out port call.

Model can have constraints on state and configuration parameters values. These constraints are defined using Scala require function.

Example of require statement for “Amplifier” component type:

```
class Amplifier...{
  class Config(val factor: Int) {
    require (factor>0 && factor<50)
  } ...
}
```

Listing 3. Configuration constraint for “Amplifier” component type. “factor” can take values only in the interval from 1 to 49.

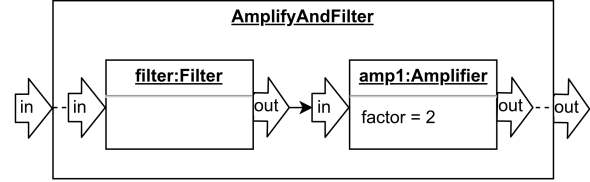


Fig. 3. “AmplifyAndFilter” component type.

2) *System Integrator View*: System integrator creates instances of component type and connects them. For each instance he defines its configuration parameters values.

As an example of component instances and their connections consider model of the scheme depicted in the Fig. 2

```
val actuator = new Actuator

val amp1 = new Amplifier(actuator.in) {
  val config = new Config(factor = 2)
}
sensor1 = new Sensor(amp1.in)

val amp2 = new Amplifier(actuator.in) {
  val config = new Config(factor = 10)
}
sensor2 = new Sensor(amp2.in)
```

Listing 4. Amplifier instances connection scheme.

3) *Preconfigured components*: There is often component which have configuration parameters that have the same value in different configuration. To simplify configuration process for system integrator, we can define new component type, in which these parameters are fixed and cannot be configured. New component type class constructor calls constructor of the original one with values of these parameters. For example, it is possible to define “AmplifierBy2” which amplifies signal by fixed factor of 2.

It is more interesting to define new component, which is a composition of existing components. This is useful if some compositions are used often. Our approach assumes unified modelling of components and their composition. This allows using component-composition transparently for system integrator.

As an example, assume that there are component type “Amplifier” and “Filter”, that are often connected. We create a new component type “AmplifyAndFilter” that is the composition of “Amplifier” and “Filter” Graphical representation of the “AmplifyAndFilter” component type can be seen at Fig. 3 and implementation:

```
class AmplifyAndFilter(out: SignalProcessor)
  extends Component {
  val amp = new Amplifier(out) {
    val config = new Config(factor)
  }
  val filter = new Filter(amp.in)

  object in extends SignalProcessor {
    def processSignal(x: Int): Int = filter.in(x)
  }
}
```

Listing 5. “AmplifyAndFilter” component type.

B. Model Usage

We use model to simulate instances behaviour and their interaction. We can verify that constraints are hold during simulation. Also, we can write tests (unit and integration) to check that component model is correct.

We use model to generate C code, which gets into JetOS. We statically parse Scala code, extract needed information and translate it into C code.

Generated C code structurally looks much like code generated by prototype based on YAML files. We use same approach to model OOP in C language.

Some parts of the model can be translated into C without modifications, for example, simple operations and function calls. Some parts modified automatically during translation, but some can not be automatically translated without human help.

JetOS has strict coding style and, for instance, function can not have more than one `return` statement. We can generate code according this code style and, for example, we can automatically substitute several return statements in the model with a single one in the generated code.

As was mentioned, there are also statements, which cannot be easily translated into C. Also there are situations when generator tool cannot get enough information statically analysing Scala code. To solve these problems we add annotations to Scala code. Annotations does not change behaviour of model, they used only to provide additional information for the generator tool.

We use annotations to highlight input and output ports and their type interfaces. Annotations are “inport”, “outport” and “interface” for input ports, output ports and port types respectively. As an example, “Amplifier” component type with annotations:

```
@interface
trait SignalProcessor {
  def processSignal(s: Int): Int
}

class Amplifier(@outport out: SignalProcessor)...{
  ...
  @inport
  object in extends SignalProcessor {
    def processSignal(s: Int): Int = {
      val processed = process(s)
      out.processSignal(processed)
    }
  }
}
```

Listing 6. Port type `SignalProcessor` and ports of “Amplifier” component type with annotations.

Scala language has rich syntax and not every statement can be easily translated to C. We allow annotating blocks of Scala code or Scala functions with C code. Partial example:

```
@C_code(code="int process(int* array) {...}")
def process(lst: List[Int]) = {...}
```

Listing 7. C_code annotation example.

This `C_code` annotation allows iteratively develop generator tool. At start, when tool supports only a few Scala statements, almost all code has C annotations. When support for new Scala statements adds to the tool, C annotations for these statements are no longer needed. Therefore, during tool development number of `C_code` annotations decreases.

VI. FUTURE WORK

First of all we still do not support many Scala statements and have a lot of `C_code` in our models. We are going to fix this in the new versions of generator tool.

For now system developer should write Scala code by hand. This Scala code is very simple and match a simple pattern. Thus, we can generate this Scala code from some GUI interface. Configuration constraints of the model can be extracted and added to this tool. This is one of optional future work.

Furthermore, formal model is a powerful tool and allows much more than C code generation. Formal model can be used for model checking and formal verifying internal consistency, preconditions or state invariants.

Tests and requirements can be generated based on the model and requirement generation is our next task. Requirement is the most important part of safety-critical system certification. Requirement writing is a hard hand work and automation (at least partial) will be very helpful.

VII. CONCLUSION

The paper presents continuation of the work on modularity of RTOS. OS drivers are decomposed into isolated components. Component composition is carried out by system integrator, and it can be done without contacting component developers and without writing C code.

We use a unified formal model to specify both components and their composition. Model, which is written in Scala language, is used to generated C code.

Also, model is executable, this allows system integrator to quickly verify correctness of composition. Model contains constraints on the model parameters. These constraints are tested during model simulation, also constraints can be translated into asserts in the generated C code.

Model-based approach still has disadvantage since the model is divided in two parts written in two languages, which have to be manually kept consistent. However, C code for some Scala statement is placed right before the statement, we hope that this will stimulate developers to update parts synchronously. Maturing of the generator tool decreases amount of C code in the model and reduces the importance of the problem.

The approach has been successfully tested on OS drivers of JetOS — ARINC-653 compliant RTOS. ARINC-653 has restrictions on the code executed in OS. For instance, resources (like buffers, semaphores, threads, etc.) can be requested only during initialization stage. Model restriction on threads creation apply well to ARINC-653 restrictions. Moreover, constructor code of the component type class is executed during

initialization stage. Thus, component can request resources in the constructor.

REFERENCES

- [1] K. M. Mallachiev, N. V. Pakulin, and A. V. Khoroshilov, "Design and architecture of real-time operating system," *Proceedings of the Institute for System Programming of the RAS*, vol. 28, no. 2, pp. 181–192, 2016.
- [2] J. Siegel and D. Frantz, *CORBA 3 fundamentals and programming*. John Wiley & Sons New York, NY, USA:, 2000, vol. 2.
- [3] N. Wang, D. C. Schmidt, and C. O’Ryan, "Overview of the corba component model," in *Component-Based Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 557–571.
- [4] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther, "The sawmill multiserver approach," in *Proceedings of the 9th workshop on ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*. ACM, 2000, pp. 109–114.
- [5] I. Boule, M. Gien, and M. Guillemont, "Chorus distributed operating systems," *Computing Systems*, vol. 1, no. 4, 1988.
- [6] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The flux oskit: A substrate for kernel and language research," in *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5. ACM, 1997, pp. 38–51.
- [7] A. Massa, *Embedded software development with eCos*. Prentice Hall Professional Technical Reference, 2002.
- [8] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller, "Think: A software framework for component-based operating system kernels," in *USENIX Annual Technical Conference, General Track*, 2002, pp. 73–86.
- [9] F. Loiret, J. Navas, J.-P. Babau, and O. Lobry, "Component-based real-time operating system for embedded applications," in *International Symposium on Component-Based Software Engineering*. Springer, 2009, pp. 209–226.
- [10] K. Mallachiev, N. Pakulin, A. Khoroshilov, and D. Buzdalov, "Using modularization in embedded os," *Proceedings of the Institute for System Programming of the RAS*, vol. 29, no. 4, pp. 283–294, 2017.

STATIC VERIFICATION FOR MEMORY SAFETY OF LINUX KERNEL DRIVERS*

Anton Vasilyev

Ivannikov Institute for System Programming, RAS
Moscow, 109004, 25, A. Solzhenitsyn st.

vasilyev@ispras.ru

Abstract—Memory errors in the Linux kernel drivers are a kind of serious bugs that could lead to dangerous consequences but such errors are hard to detect. Debug and disclose of kernel memory corruptions often requires specially compiled kernel.

Static verification of industrial projects such as Linux kernel requires additional effort to support inaccuracy produced by environment model. The current work proposes an approach to reveal issues with memory usage in incomplete parts of ANSI C programs. Our static verification technique based on Symbolic Memory Graphs (SMG) theory with extensions aims to reduce rate of false alarms. Methods were evaluated within Linux Driver Verification project.

Index Terms—shape analysis, static verification, symbolic memory graphs, memory model

I. INTRODUCTION

Operating system kernel is often written in the C programming language. This language is portable and effective, but unfortunately it is not memory safety. Issues in source code could lead to vulnerability or unpredictable failures. Common methods such as testing are unable to find all problems. A probable solution to get an evidence of source code properties is formal methods and there are results of comprehensive formal verification of the seL4 microkernel [1]. However formal methods generally require a closure for a verification object and its environment to produce an appropriate verdict. This article considers operating system kernel drivers with automatically generated environment models as a target for approbation of a memory verification technology.

Main contributions of this paper are connected with extensions of an existed static memory verification approach to be able to perform Linux kernel drivers verification, which are described in section V.

II. LINUX DRIVER VERIFICATION

The Linux kernel represents an industrial code base with more than 10 million lines of drivers code. A distinctive feature of Linux is instability of internal interfaces. A high speed of changes with a distributed development process requires an efficient bug finding strategy.

The research of typical faults in Linux operating system drivers divides errors into typical and atypical [2]. Atypical faults in drivers are described as connected with specific hardware and not applicable to other drivers. Typical faults can be specified by some rule which is true for all or some group of drivers. Typical faults are further divided into:

- Linux specific faults, which correspond to kernel interfaces usage rules.
- Races and deadlocks, which are related with parallel execution.
- Generic problems, which are common for C programs such as null pointer dereference, integer overflow, etc.

Authors show that 29.2% of typical errors fixed in stable branches of the Linux kernel are generic problems. Statistics about memory problems corresponding to all generic faults is shown in the table:

| Type | Percentage |
|-----------------------------|------------|
| null pointer dereference | 30.4% |
| resource: | 23.5% |
| memory leak, | |
| double free, | |
| use after free | |
| buffer overflow | 7.8% |
| uninit: | 5.9% |
| uninitialized pointer free, | |
| write to unallocated memory | |
| Total | 67.6% |

This information shows that main part of stable kernel fixes of generic faults match memory errors. We suggest to improve situation with memory safety of stable Linux kernel with help of static verification.

Linux Driver Verification project (LDV) [3], [4], [5] aims at performing automatic static driver verification and reporting detected problems. It provides a static verification framework called *Klever* [6] for Linux kernel verification including automated environment model generation [7], [8], rules of correct kernel API usage [2], interfaces for storing and visualization of verification results [9]. As a verification engine *Klever* includes the CPAchecker [10] verification tool.

In this work we have made the following contributions:

- 1) improved *Klever* environment models to remove memory errors inside them
- 2) added several Linux kernel specific extensions into CPAchecker verification tool for memory safety verification.

We have made experimental evaluation on drivers of Linux kernel v.4.11.6, analyzed all memory safety problems reported by the verification tool and classified them into bugs and false alarms. We have prepared bug reports and fixes to the newest kernel version at the moment. Regarding false alarms we

*The research was supported by RFBR grant 18-01-00426

conclude that automatic generation heavily affects verification results and requires further improvement.

III. SHAPE ANALYSIS

Shape analysis is a verification method with abstractions allowing to describe dynamic structures such as heap allocated data, linked lists, hash tables, etc [11]. A central function used to extract an abstraction from a memory state is a summarization procedure. Materialization is an opposite operation which slices an abstraction to concrete items.

We describe shape analysis which can disclose following types of memory issues:

- memory leak,
- double free,
- uninitialized pointer free,
- write to unallocated memory,
- use after free,
- buffer over read/write,
- null pointer dereference.

IV. SYMBOLIC MEMORY GRAPHS

The symbolic memory graph (SMG) algorithm [12] is a kind of shape analysis. It works with directional graph representation of memory state. Nodes are used for symbolic values, memory regions and abstracted structures representation. Edges show references between nodes and are divided into *point-to edges* for pointers and *has-value edges*. Each edge and node in SMG has a set of *labels* representing size, offset, allocation status. One symbolic memory graph with abstractions can represent several memory states called concrete memory images. Set of all concrete memory images for SMG G is denoted as $MI(G)$.

Our SMG implementation in CPAchecker [10] keeps mapping between global, stack variables and memory regions. Also it tracks correspondence between symbolic and concrete values. A memory graph is modified in correspondence with instructions from source code.

Detailed description of operations on SMG can be found at [12]. Here we provide a brief overview.

- *Data Reinterpretation*
 - *Read Reinterpretation*
 - *Write Reinterpretation*

Modifications. A level of details for memory model allows to take into account such low level interpretation as unions and provide facility for reinterpretation values even on the same offset with different types.

Algorithm supports partial values overwrite if memory for corresponding field intersects. For example:

```
1 union {
2     int i;
3     char c;
4 } u;
5 u.i = 10;
6 u.c = 'A';
```

After line 5 union u will contain integer value 10 with size 4 byte, but after line 6 from this union we are able to read 1 byte char 'A' or undefined 4 byte integer value.

Checks. For these operations the tool performs checking for read/write within object bounds against null pointer dereference, buffer overread/overwrite errors.

- *Join of SMGs*

This operation is central one for abstraction and decision whether current memory state is covered by another and vice versa, so the algorithm can drop one of the states. It takes as input 2 SMGs G_1, G_2 , compares their concrete memory images and produces join status with summarization SMG G . If $MI(G_1) \not\subseteq MI(G_2)$ and $MI(G_1) \not\supseteq MI(G_2)$ then SMGs are semantically incomparable and their join is undefined. Algorithm travels through pair of source SMGs and tries to join nodes. It is possible if nodes have same size, validity, and special conditions for join with abstract lists. Abstract lists are join-able if they have same head, prev and next offsets, join result will have number of elements equal to minimum from originals. Also result of join region with abstract list become abstract list. It is possible to insert empty list abstraction at any correct position in graph to increase opportunity of correct join.

- *Summarize sequence of objects to list abstraction*

Algorithm discovers sequences of neighboring objects which could be considered as candidate list entries and then sequentially adds them into one abstract list and increases its size.

- *Abstract list materialization*

Opposite operation to the summarize.

- *Checking equality and inequality of values and pointers*

Algorithm performs sound and efficient but incomplete check for equality and inequality of values. In some cases it can fail on abstractions.

Tool performs stack variables cleaning on function exit and checking for dangling pointers to allocated memory, which helps identify memory leak errors.

Let's consider analysis of a simple example:

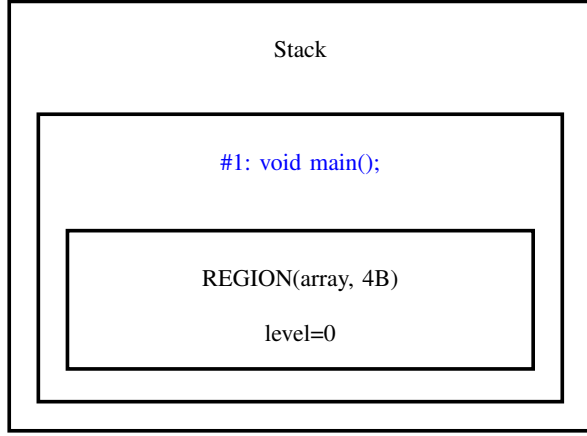
```
void main() {
1   void *array;
2   long b = 2;
3   long c = 3;
4   array = calloc(1, 16);
5   memcpy(&array[4], &b, 4);
6   memcpy(&array[5], &c, 4);
}
```

Fig. 1:

- Modification: allocate 4 byte memory region on stack for a pointer array.

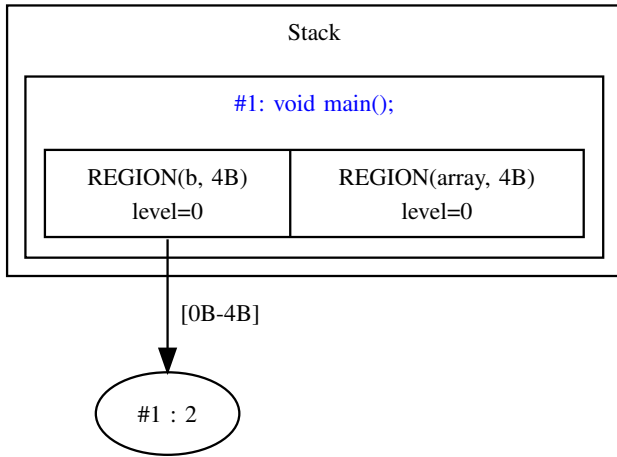
Fig. 2:

- Modification: allocate 4 byte memory region on stack for long b and assign new value #1 with explicit value 2L.
- Check: memory size is sufficient for assigned value.



Location: 1 void *array;

Fig. 1



Location: 2 long b = 2;

Fig. 2

Fig. 3:

- Modification: allocate 4 byte memory region on stack for long c and assign new value #2 with explicit value 3L.
- Check: memory size is sufficient for assigned value.

Fig. 4:

- Modification: allocate 16 byte memory region on heap (mark it by tag calloc_ID3), fill it by NULL values, and assign array a new point-to-value #4 which points to 0 offset of region calloc_ID3.
- Check: memory size is sufficient for assigned values.

Fig. 5:

- Modification: assign 4 byte value #1 by offset 4 of region calloc_ID3.

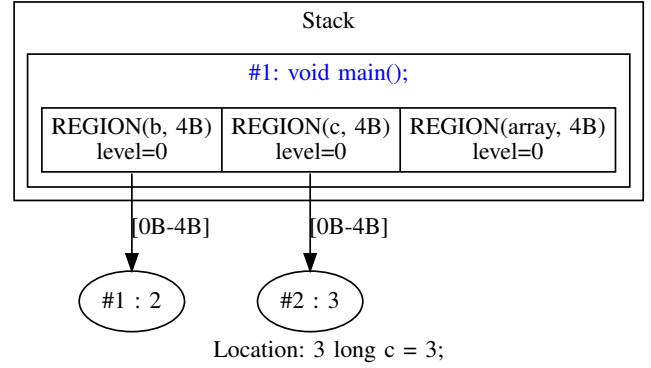
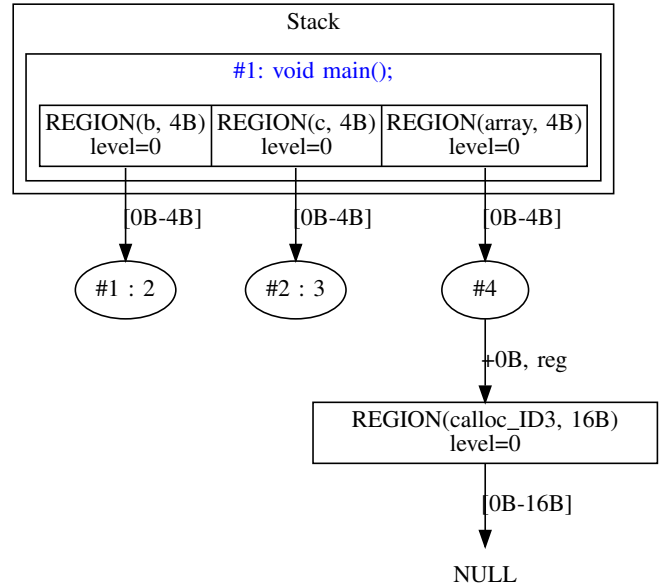


Fig. 3



Location: 4 array = calloc(1, 16);

Fig. 4

- Check: dereference and assignment are done within allocated memory.

Fig. 6:

- Modification: assign 4 byte value #2 by offset 5 of region calloc_ID3, remove intersecting values, so value at offset 4 of region calloc_ID3 is not defined.
- Check: dereference and assignment are done within allocated memory.

V. LINUX SPECIFIC EXTENSIONS FOR SMG

A. Bit precise model

Linux kernel code operates on structures with bit fields. We have support bit fields in CPAChecker [10] and switched SMG operations granularity from byte to bit precision.

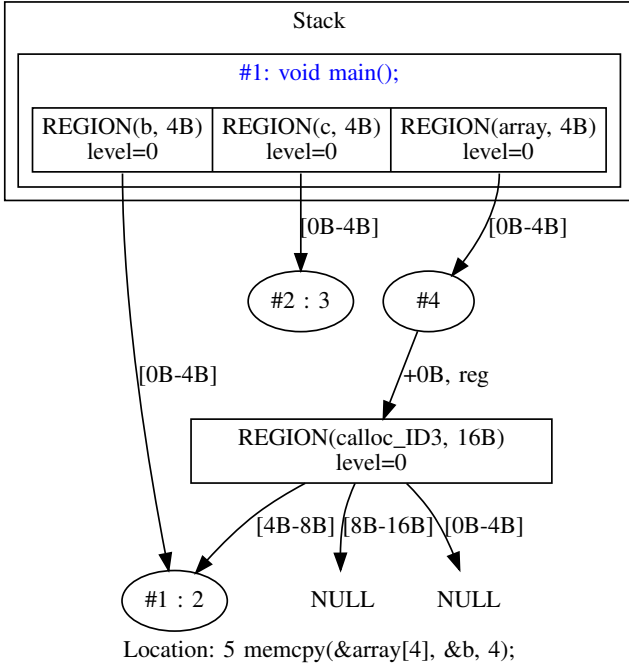


Fig. 5

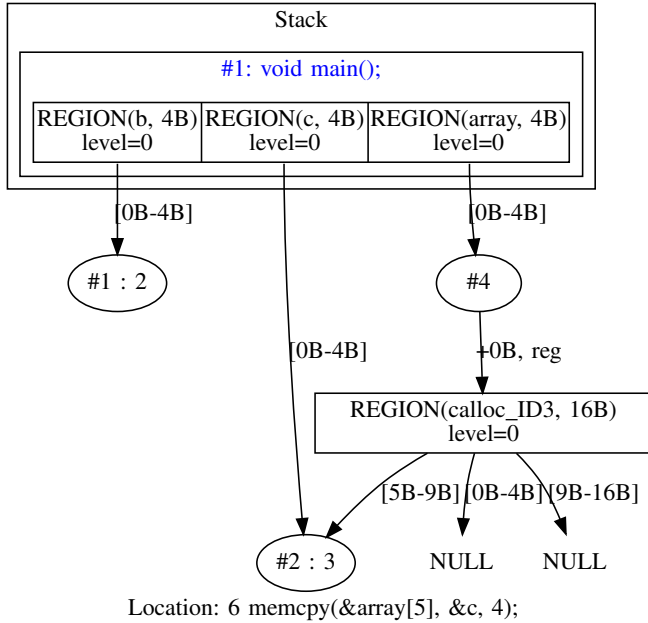


Fig. 6

B. Predicate extension

Complexity of drivers code requests us to make precise conditions tracking for filtering infeasible paths. We implemented this feature by enriching SMG state with predicate path annotation on symbolic and concrete SMG values. On branching we perform predicate satisfiability check to decide which branch is feasible. In addition, this method allows us

to extend memory region over-read and overwrite checks for arrays using error predicate check on data reinterpretation operation.

C. Memory on demand

For simplification we consider Linux kernel as trusted code and drivers as untrusted code in following sense: all structures provided to drivers on initialization by the kernel core are controlled by the kernel. We assume that the kernel recursively initializes all fields so drivers do not require to manage these structures. We support current point of view as a memory on demand concept within CPAchecker.

Memory on demand is marked by special function `void* ext_allocation()`. Returned pointer allows any recursive dereference by any offset and distinguishes values by list of offsets and pointers from original pointer. Additionally any explicitly allocated memory which is reachable from memory on demand is considered as automatically freed on program exit. Let's illustrate concept on a simple example:

```

1  struct S {
2      int i;
3      struct S *s;
4  } *top;
5  top = ext_allocation();
6  top->i = 5;
7  top->s->s = malloc(8);
8  top->s->i = top->i;
9  return;

```

- Line 5: Mark top as *memory on demand*
- Line 6: Store 5 at `top->i`
- Line 7: Recursively initialize pointer `top->s` as *memory on demand* block and store pointer to new memory block at `top->s->s`
- Line 8: Read value from `top->i` and store it at `top->s->i`
- Line 9: Check of memory leaks excludes `top`, `top->s` and `top->s->s`.

This example shows possible way for environment simplification to reduce false positive rate of memory errors.

VI. CONFIGURABLE PROGRAM ANALYSIS

Theory of SMG is implemented as Configurable Program Analysis (CPA) [13] within CPAchecker tool [10] under the name SMGCPA.

Common CPA has *abstract domain*, *transfer*, *merge* and *stop* operators.

- *Abstract domain* describes abstract states which represent sets of concrete states of the program.
- *Transfer* operator gets one state and a control flow operation as input and returns all states which appears after applying the operation on the original state.
- *Merge* operator takes 2 states as input and tries to combine them into one.

TABLE I: Evaluation on Linux kernel drivers v.4.11.6

| | | | |
|---------|------|---------------|------|
| Safe | 1560 | | |
| Unknown | 4023 | Timeouts | 2594 |
| | | Others | 1429 |
| Unsafe | 641 | Bugs | 49 |
| | | False alarms | 512 |
| | | Without marks | 80 |

- *Stop* operator identifies when one state is covered by others and decides whether it is required to continue analysis with a current state.

CPAchecker allows to combine different CPAs into one composite CPA. It works with a composite state which includes states of each involved CPAs. *Merge* operator produces a Cartesian product of separate analyses *merge* results.

SMGCPA fits into CPA conception with the following operators:

- *Abstract domain* has SMG states as abstractions.
- *Transfer* performs SMG transformations corresponding to current control flow operation.
- *Merge* tries to join SMGs from states and returns new SMG if join is successful.
- *Stop* checks whether $MI(G_1) \subseteq MI(G_2)$ or a state has memory issues.

VII. EXPERIMENTAL RESULTS

Experiments were performed with the help of *Klever* static verification framework [6] a part of LDV project [4].

Klever automatically generates environment model and a general use case driver life-cycle for each separate driver. We performed verification of Linux kernel drivers v.4.11.6 on memory safety.

See Table I for results of an experiment on 6224 generated verification tasks with 15 minutes time limit for each task. We performed manual analysis of 561 Unsafe verdicts and classified 49 Unsafes as real memory bugs and 512 as false alarms.

The causes of 512 false alarms are the following:

- Imprecise environment models (258).
Automatically generated environment models could mistakenly provide wrong driver initialization and deallocation. Also some emulated functions are imprecise for correct proof of memory safety.
- Absent function (139).
Current environment models do not contain functions imported from other drivers. This leads to false positive verdicts if missed functions are important for memory safety properties.
- Require predicate SMG (83).
These false alarms are connected mainly with arithmetic operations on unknown values. We expect that some common patterns used in software could be emulated by additional predicates description, e.g. bitwise AND on unsigned values provide result value less or equal to

operands and this is common check for array dereference in kernel.

- SMG problems (13).
Problems with analysis such as missed values after merge and wrong assumption about loop invariants.
- Task generator problems (10).
Current task generator omits information about packed pragma for structures at final file, but on preprocessing step this information is available and may be inlined at `alloc(sizeof(...))` constructions. Sometimes it provides less allocation size then unpacked structure size.
- Unknown allocation sizes (9).
If SMG cannot derive explicit value for allocation size it uses predefined value, which may be less then required.

The list of reported bugs is presented in Table II. Not all bugs were reported, because some of them were detected in old unsupported drivers or were already fixed.

Let's consider the bug 2017/8/15/322 from Table II discovered in Samsung I2S Controller driver within Linux kernel v.4.11.6 for which our patch was applied in v.4.14-rc1.

Klever provides full error trace from entry point to error occurrence for unsafe verdict on the left side of screen. Right side is used for source code relevant to selected trace line. The parts of the trace for I2S driver are shown in figure 7.

Fig. 7a shows a part of the error trace with the declaration of the structure `struct i2s_dai *pri_dai` in function `samsung_i2s_probe()`. In the same function in Fig. 7b `pri_dai` is initialized by function `i2s_alloc_dai()` (line 1246), and field `sec_dai` becomes NULL (line 1095).

The third part of the error trace in Fig. 7c shows that `sec_dai` initialization is skipped by condition in line 1319 (`quirks & QUIRK_SEC_DAI`) triggered by device capabilities, so `pri_dai` is remained equal to NULL.

In the same figure we see that the structure `pri_dai` becomes stored at `driver_data` by `dev_set_drvdata()` in line 1363 and then extracted by `dev_get_drvdata()` in line 1382 of `samsung_i2s_remove()`. Next we assign `sec_dai` in line 1383 and then perform dereference of `sec_dai` in line 1386 without check for NULL, which leads to *Null pointer dereference*.

The bug could be reproduced on Samsung "s3c6410-i2s" and "exynos7-i2s1" devices by inserting and removing the driver module `sound/soc/samsung/i2s.ko`, because the condition in line 1319 is false for `i2sv3_dai_type` and `i2sv5_dai_type_i2s1` (see lines 1454 and 1477 in `sound/soc/samsung/i2s.c`).

VIII. RELATED WORK

Linux kernel is important open source software. Many research and industrial projects improve kernel quality by testing, bug hunting, fuzzing or error reports. There are available examples: The Linux Test Project ¹, Autotest ², Kmemleak, Kmemcheck, kselftest, Fault Injection Framework.

¹<http://linux-test-project.github.io>

²<http://autotest.github.io>

TABLE II: Bugs reported to Linux Kernel Mailing List (<https://lkml.org/lkml>)

| Message ID | Subject |
|---------------|--|
| 2017/8/1/615 | Buffer overread in pv88090-regulator.ko |
| 2017/8/10/693 | hwmon:(stts751) buffer overread on wrong chip |
| 2017/8/10/597 | dmaengine: qcom_hidma: avoid freeing an uninitialized pointer |
| 2017/8/15/322 | ASoC: samsung: i2s: Null pointer dereference on samsung_i2s_remove |
| 2017/8/10/535 | i2c: use release_mem_region instead of release_resource |
| 2017/8/16/493 | mtd: plat-ram: Replace manual resource management by devm |
| 2017/8/11/366 | mISDN: Fix null pointer dereference at mISDN_FsmNew |
| 2017/8/10/522 | parport: use release_mem_region instead of release_resource |
| 2017/8/11/368 | video: fbdev: udlfb: Fix use after free on dlfb_usb_probe error path |
| 2017/8/10/550 | dvb-usb: Add memory free on error path in dw2102_probe() |
| 2017/8/16/345 | udc: Memory leak on error path and use after free |

CPAchecker has ability to disclosure race and reachability errors. These error classes are used by LDV project as checks for correct interface usage at Linux kernel and races.

Microsoft research has a project called Angelic Verification [14] which aim is to provide static assertion checking in open programs with low level of false-positives.

IX. CONCLUSIONS AND FUTURE WORK

We have presented approach to static verification of Linux kernel code on memory errors. We expect to reduce SMG false positive rate by introducing precise predicate extension. Further efforts will be aimed at reducing timeouts and new efficient abstractions for arrays and multiple values support. Also current environment models should be improved to reduce number of false positive verdicts.

REFERENCES

- [1] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014. [Online]. Available: <http://doi.acm.org/10.1145/2560537>
- [2] V. Mutilin, E. Novikov, and A. Khoroshilov, "Analiz tipovyh oshibok v drajverah operacionnoj sistemy Linux (Analysis of typical faults in Linux operating system drivers) (in Russian)," *Proceedings of the Institute for System Programming of RAS*, vol. 22, pp. 349–374, 2012. [Online]. Available: <http://doi.org/10.15514/ISPRAS-2012-22-19>
- [3] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, "Establishing Linux driver verification process," in *Perspectives of Systems Informatics*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 5947, pp. 165–176.
- [4] Web-site, "Linux driver verification project," <http://linuxtesting.org/ldv>.
- [5] I. Zakharov, M. Mandrykin, V. Mutilin, E. Novikov, A. Petrenko, and A. Khoroshilov, "Configurable toolset for static verification of operating systems kernel modules," *Programming and Computer Software*, vol. 41, no. 1, pp. 49–64, 2015. [Online]. Available: <http://dx.doi.org/10.1134/S0361768815010065>
- [6] Web-site, "Klever verification framework," <https://forge.ispras.ru/projects/klever>.
- [7] I. S. Zakharov, V. S. Mutilin, and A. V. Khoroshilov, "Pattern-based environment modeling for static verification of linux kernel modules," *Program. Comput. Softw.*, vol. 41, no. 3, pp. 183–195, May 2015. [Online]. Available: <http://dx.doi.org/10.1134/S036176881503007X>
- [8] A. Khoroshilov, V. Mutilin, E. Novikov, and I. Zakharov, "Modeling environment for static verification of linux kernel modules," in *Perspectives of System Informatics*, A. Voronkov and I. Virbitskaite, Eds., vol. 8974. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 400–414. [Online]. Available: https://doi.org/10.1007/978-3-662-46823-4_32
- [9] E. Novikov and I. Zakharov, "Towards automated static verification of GNU C programs," in *Perspectives of System Informatics*, A. K. Petrenko and A. Voronkov, Eds., vol. 10742. Springer International Publishing, 2018, pp. 402–416. [Online]. Available: https://doi.org/10.1007/978-3-319-74313-4_30
- [10] D. Beyer and M. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6806, pp. 184–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_16
- [11] R. Wilhelm, S. Sagiv, and T. W. Reps, "Shape analysis," in *Proceedings of the 9th International Conference on Compiler Construction*, ser. CC '00. London, UK, UK: Springer-Verlag, 2000, pp. 1–17. [Online]. Available: <http://dl.acm.org/citation.cfm?id=647476.760384>
- [12] K. Dudka, P. Peringer, and T. Vojnar, "Byte-precise verification of low-level list manipulation," in *Static Analysis*, F. Logozzo and M. Fähndrich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 215–237. [Online]. Available: https://doi.org/10.1007/978-3-642-38856-9_13
- [13] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: concretizing the convergence of model checking and program analysis," in *Proceedings of CAV*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 504–518. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1770351.1770419>
- [14] A. Das, S. K. Lahiri, A. Lal, and Y. Li, "Angelic verification: Precise verification modulo unknowns," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 324–342.

Fig. 7: Error trace for I2S driver

(a) probe function

```

1231 tmp = samsung_i2s_probe(arg1);
1231   ↳ ldv_2_probed_default = samsung_i2s_probe(ldv_2_resource_platform_device, &ldv_2_probed_default);
1231 struct i2s_dai *pri_dai;
1229 static int samsung_i2s_probe(struct platform_device *pdev)
1230 {
1231     struct i2s_dai *pri_dai, *sec_dai = NULL;

```

(b) pri_dai initialization

```

1246   ↳ pri_dai = i2s_alloc_dai(pdev, 0);
1087 struct i2s_dai *i2s;
1089 i2s = (struct i2s_dai *)tmp;
1090 assume(((unsigned long)i2s) != ((unsigned long)((struct i2s_dai *)0));
1093 i2s->pdev = pdev;
1097 i2s->pri_dai = (struct i2s_dai *)0;
1094 i2s->sec_dai = (struct i2s_dai *)0;
1096 i2s->i2s_dai_drv.symmetric_rates = 1U;
1097 i2s->i2s_dai_drv.probe = &samsung_i2s_dai_probe;
1098 i2s->i2s_dai_drv.remove = &samsung_i2s_dai_remove;
1099 i2s->i2s_dai_drv.ops = &samsung_i2s_dai_ops;
1100 i2s->i2s_dai_drv.suspend = &i2s_suspend;
1084 static struct i2s_dai *i2s_alloc_dai(struct platform_device *pdev)
1085 {
1086     struct i2s_dai *i2s;
1087     i2s = devm_kzalloc(&pdev->dev, sizeof(struct i2s_dai), GFP_KERNEL);
1088     if (i2s == NULL)
1089         return NULL;
1090     i2s->pdev = pdev;
1091     i2s->pri_dai = NULL;
1092     i2s->sec_dai = NULL;

```

(c) dev_set_drvdata/dev_get_drvdata and Null pointer dereference

```

1316 assume(ret == 0);
1319 assume((quirks &2U) == 0U);
1357 assume(((unsigned long)i2s_pdata) == ((unsigned long)((struct i2s_pdata *)0));
1363 dev_set_drvdata(&pdev->dev, (void *)pri_dai);
1363   ↳ dev_set_drvdata(&pdev->dev, (void *)pri_dai);
1033 dev->driver_data = data;
1034 return;
1365   ↳ pm_runtime_set_active(&pdev->dev);
1366 pm_runtime_enable(&pdev->dev);
1368   ↳ ret = i2s_register_clock_provider(pdev);
1369 assume(ret == 0);
1370 return 0;
361   ↳ ldv_2_probed_default = ldv_post_probe(ldv_2_probed_default);
438 Remove device from the system. Invoke callback remove from platform
1380 samsung_i2s_remove(arg1);
1380   ↳ samsung_i2s_remove(ldv_2_resource_platform_device);
1380 struct i2s_dai *pri_dai;
1381 struct i2s_dai *sec_dai;
1382 pri_dai = (struct i2s_dai *)tmp;
1382   ↳ pri_dai = (struct i2s_dai *)dev_get_drvdata((struct device *)pdev);
1028 return ((void *)dev->driver_data);
1028   ↳ return ((void *)dev->driver_data);
1383 sec_dai = pri_dai->sec_dai;
1385 pri_dai->sec_dai = (struct i2s_dai *)0;
1386 NULL pointer dereference on write
1386 sec_dai->pri_dai = (struct i2s_dai *)0;
1318 if (quirks & QUIRK_SEC_DAI) {
1319     sec_dai = i2s_alloc_dai(pdev, true);
1320     if (!sec_dai) {
1321         dev_err(&pdev->dev, "Unable to alloc I2S_sec\n");
1322         ret = -ENOMEM;
1323         goto err_disable_clk;
1324     }
1325     sec_dai->lock = &pri_dai->spinlock;
1326     sec_dai->variant_regs = pri_dai->variant_regs;
1327     sec_dai->dma_playback.addr = regs_base + I2STXDS;
1328     sec_dai->dma_playback.chan_name = "tx-sec";
1329     if (!np) {
1330         sec_dai->dma_playback.filter_data = i2s_pdata->dma_filter;
1331         sec_dai->filter = i2s_pdata->dma_filter;
1332     }
1333     sec_dai->dma_playback.addr_width = 4;
1334     sec_dai->addr = pri_dai->addr;
1335     sec_dai->clk = pri_dai->clk;
1336     sec_dai->quirks = quirks;
1337     sec_dai->idma_playback.addr = idma_addr;
1338     sec_dai->pri_dai = pri_dai;
1339     pri_dai->sec_dai = sec_dai;
1340     ret = samsung_asoc_dma_platform_register(&pdev->dev, sec_dai->filter, "tx-sec", NULL);
1341     if (ret < 0)
1342         goto err_disable_clk;

```

Configurable system call tracer in Qemu emulator

Ivanov Alexey

Novgorod State University

Russia, Velikiy Novgorod, B. St-Peterburgskaya, 41

alexey.ivanov@ispras.ru

ABSTRACT

The paper describes a method for tracing applications through configurable system calls in a virtual machine. This method is based on plugins, receiving information about system calls from the running system and outputs it to a file. The plugin loads a configuration file that corresponds to the running operating system. The plugin parses the information received and, based on it, traces the OS and applications.

KEYWORDS

Qemu, configurable system calls, debugging, plugin, system calls, tracing.

1 INTRODUCTION

Sometimes programmers face the task of analyzing the work of a compiled program to find its flaws, defects, and even search for malicious code in it. To analyze the work of such applications, we have to study their binary code or try to decompile the code, which is a laborious task. In order to simplify the analysis of applications, we can use the system calls of this application. System calls provide an essential interface between a program and the operating system. It is possible to track which system calls the application makes, and draw conclusions about the behavior of the program. This method allows us to debug the application without delving into the level of instructions and architecture features, thereby reducing the time required to find the problem.

Debugging applications using system tracing can be done inside the operating system, but a number of problems arise: strong dependence of the debugger on the operating system; it is not possible to run several debuggers at the same time; there is no access to the privileged execution; it is necessary to secure the operating system when analyzing programs that have harmful effects. To solve these problems, we can use the virtual machine tools. In this way, we can debug applications in a wide range of different operating systems running under different process architectures.

2 APPROACH AND UNIQUENESS

To date, several debuggers allow us to trace an application using system calls. All these debuggers have a drawback: they do not provide enough portability of the debugger between different operating systems and processor architectures. We offer a new

approach to implementing the debugger through system calls, load all the information necessary for tracing from the configuration file. The configuration files will allow us to easily configure and change the parameters needed for debugging, and also simplify the addition of support for new operating systems and architectures without recompiling the program and learning the debugger code.

It was decided to implement the debugger under the virtual machine QEMU[1], using the plugin mechanism. QEMU is an open source virtual machine that emulates the hardware of various platforms. This virtual machine supports the emulation of a large number of processors such as x86, PowerPC, ARM, MIPS, SPARC, m68k. Also, this simulator supports the launch of a large number of different operating systems.

At the moment for QEMU there is a plugin mechanism implemented by ISP RAS[2], which allows us to connect developed plugins to a virtual machine both during startup and during its operation. In the plugin mechanism an implemented, functional allows each additional translation of the instruction to substitute additional code for execution, when this instruction is called. This mechanism is suitable for debugging through system calls, so it was decided to use it.

In addition, various mechanisms of the system call play an important role. The classical way of implementation is the use of interrupts. With the help of interrupts, control is transferred to the kernel of the operating system, with the application having to enter the number of the system call and the necessary arguments into the corresponding registers of the processor.

For many RISC processors, this method is the only one, however, the CISC architecture has additional methods. The two mechanisms developed by AMD and Intel are independent of each other, but, in fact, perform the same functions. These are SYSCALL / SYSRET or SYSENTER / SYSEXIT statements. They allow us to transfer control to the operating system without interrupts.

Each operating system supports values returned from the system call, which are passed as reference types when the system call handler is called. During the execution of the system call, the service procedure records the required values if necessary by the available links, after which the system call is exited.

One of the main tasks that we had to face was the task of supporting the plugin of different operating systems and processor architectures. The solution to this problem was the interface with the configuration file. The configuration file makes the debugger more flexible and customizable. With its help, we can disconnect

from the trace a certain mechanism of system calls or disable unnecessary system calls. In addition, such a mechanism makes it easier to add support for new operating systems and processor architectures.

To implement the interface with the configuration file, it was necessary to study a wide range of different operating systems and processor architectures. After gathering the necessary information, we can determine the information necessary for debugging: what type of system call is supported by SYSCALL / SYSRET or SYSENTER / SYSEXIT and their opcodes; location of system call arguments; a list of system calls, with the name of each system call, its code, and the list of arguments. Thus, by developing an interface for debugger and configuration file interfacing, we can add support for operating systems without going into the debugger code.

When implementing the debugger interaction interface with the configuration file, it became necessary to recognize the various expressions read from the file. For this task, we used the generator of the bison parser and developed the corresponding grammar[3].

3 BACKGROUND AND RELATED WORK

At the moment, there are several debuggers to solve existing problems. Nitro[4] allows us to trace system calls, but it works only under Intel x86 architecture. Another debugger - Panda[5], can also trace system calls, supports operating systems: Linux, Windows 7, Windows XP and two architectures of the i386 processor and ARM. The description of all system calls is found in the code of this debugger, as a result of which this approach makes it difficult to add support for new operating systems and processor architectures, and also worsens the flexibility in configuring the plugin, since the system debugger settings mechanism is not provided.

4 CONCLUSION AND DISCUSSION

Based on the results of the work done, a plugin was developed in the QEMU virtual machine, with which we can trace and debug an application using system calls. As input to the plugin, the configuration file corresponding to the operating system running in the QEMU virtual machine and corresponding to the selected processor architecture is used.

The structure of the configuration file consists of 4 parts. The first part provides information about the operating system, name and bit capacity. The second part is responsible for the supported mechanisms of system calls. The next part contains the location of the system call arguments. The last part contains a list of all available system calls and service information about the arguments of the system call.

As result of the plugin's work a log file is created that contains all the system calls that the plugin has intercepted. Each system call displays detailed information: the name and value of each system call argument, the number of the thread of execution from which this system call was made, and the value that returned the system call after execution. In Fig. 1 presents a small fragment of the output file that was created by the implemented plugin running

in the windows XP operating system and the i386 processor architecture.

```
0x3e84000 entr: 0x114: NtWriteRequestData
0x3e84000 exit: 0x114: NtWriteRequestData
return: 0x0
0x3e84000 entr: 0xc4: NtReplyWaitReceivePortEx
0x3e84000 entr: 0x112: NtWriteFile
arg 0: 0x2a4 (HANDLE FileHandle )
arg 1: 0x0 (HANDLE Event )
arg 2: 0x0 (PIO_APC_ROUTINE ApcRoutine )
arg 3: 0x0 (PVOID ApcContext )
arg 4: 0x8ff6d8 (PIO_STATUS_BLOCK IoStatusBlock )
arg 5: 0x9059f8 (PVOID Buffer )
arg 6: 0xbc (ULONG Length )
arg 7: 0x8ff6e0 (PLARGE_INTEGER ByteOffset )
arg 8: 0x0 (PULONG Key )
0x3e84000 exit: 0x112: NtWriteFile
return: 0x0
0x3e84000 entr: 0x74: NtOpenFile
arg 0: 0x8ff6c4 (PHANDLE FileHandle )
arg 1: 0x100100 (ACCESS_MASK DesiredAccess )
arg 2: 0x8ff680 (POBJECT_ATTRIBUTES ObjectAttributes )
arg 3: 0x8ff6a4 (PIO_STATUS_BLOCK IoStatusBlock )
arg 4: 0x7 (ULONG ShareAccess )
arg 5: 0x204020 (ULONG OpenOptions )
0x3e84000 exit: 0x74: NtOpenFile
return: 0x0
0x3e84000 entr: 0xe0: NtSetInformationFile
arg 0: 0x31c (HANDLE FileHandle )
arg 1: 0x8ff6a4 (PIO_STATUS_BLOCK IoStatusBlock )
arg 2: 0x8ff658 (PVOID FileInformation )
arg 3: 0x28 (ULONG Length )
arg 4: 0x4 (FILE_INFORMATION_CLASS FileInformationClass )
0x3e84000 exit: 0xe0: NtSetInformationFile
return: 0x0
0x3e84000 entr: 0x19: NtClose
arg 0: 0x31c (HANDLE Handle )
0x3e84000 exit: 0x19: NtClose
return: 0x0
```

Figure 1: Part of the output file of the plugin.

From the information gathered in the log file, we can analyze the operation of the debugged application running inside the virtual machine. The operating system load time when using the developed plugin is increased by 1.2 times compared to the time of the operating system loading without this plugin.

ACKNOWLEDGMENTS

The work was partially supported by RFBR, research project No.

REFERENCES

- [1] F. Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [2] Vasiliev I.A., Fursova N.I., Dovgaluk P.M., Klimushenkova M.A., Makarov V.A. Modules for instrumenting the executable code in QEMU simulator. Journal of Formation Security Problems. Computer Systems, 2015, 4, 195-203
- [3] GNU Bison [HTML] (<https://www.gnu.org/software/bison/>)
- [4] Nitro. [HTML] (<http://nitro.pfoh.net/index.html>)
- [5] Panda. Plugin: syscalls2. [HTML] (<https://github.com/panda-re/panda/blob/master/panda/plugins/syscalls2/USAGE.md>)

Stealth debugging of programs in Qemu emulator with WinDbg debugger

Mikhail Abakumov
Novgorod State University
Velikiy Novgorod, Russia
mikhail.abakumov@ispras.ru

When programs are analyzed for the presence of vulnerabilities and malicious code, there are situations when the program changes its behavior due to the connection of the debugger. The WinDbg debugger has the possibility of connecting to a remote debug service (KdSrv.exe) in the Windows kernel. Therefore, it is possible to connect to the guest system running in the QEMU emulator. Kernel debugging is possible only with the enabled debugging mode, may change at the same time. Our module of WinDbg debugger for QEMU is an alternative of the remote debugging service in the kernel. Thus, the debugger connects to the debugging module, not to the kernel of the operating system. The module obtains all the necessary information answering debugger requests from the QEMU emulator. At the same time for debugging there is no need to enable debugging mode in the kernel. This leads to stealth debugging. Our module supports all features of WinDbg regarding remote debugging, besides interception of events and exceptions.

QEMU, Windows, WinDbg, remote debugging, stealth debugging

Introduction

When performing a dynamic analysis of binary (executable) code, the problem arises of qualitatively isolating the code and the instrumentation on which this analysis is performed. The need for isolation is dual. On the one hand, it is necessary to limit the impact of the code being studied, since it is able to affect the state of the instrument machine, which is especially important in the study of malicious software. On the other hand, analysis tools can indirectly change the behavior of the program being studied. The most indicative are the situations when errors in working with dynamic memory and race conditions cease to appear in the debugging mode.

The search for undocumented features in a binary code encounters a similar problem. Various techniques and techniques are known [1], with the help of which malware reveals that its execution takes place in a controlled environment, and does not fulfill its objective functions. To find the debugger to be connected, check the int 3 handler and hardware debug registers, evaluate the behavior of certain API functions, and track the progress of the system time.

It is possible to divide potential sources of information, which makes it possible to identify the fact of working in a controlled environment into three disjoint groups. The first is the interaction with external, uncontrolled components, the program being studied, such as remote servers. To the same category, it is necessary to include speed checks. Successful struggle with such sources allows the mechanism of

deterministic reproduction [2]. If you write the progress of the system in advance, when debugging and analyzing it during playback, there will be no effect on the guest's state because all time characteristics are fixed during recording. The second group of sources is the discrepancy in the behavior of the equipment [3]. The implementation of virtual equipment in software emulators is not always ideal. Known inaccuracies can be used to determine the emulator in which the program runs. The third group is the analysis tools present in the runtime. This kind of facility occurs even when the debugger is running in conjunction with a virtual machine.

I. RELATED WORK

In the Qemu emulator at the moment there is only a module of the GDB debugger, which allows debugging the kernel of the system, but in itself it has relatively small functionality and does not have a GUI. You can use IDA Pro Disassembler [4] to connect to the emulator via the GDB interface, but this will not extend the range of the GDB's features, but will only increase the ease of use. In addition, there is a utility called Winbagility [5], which allows the debugger WinDbg to connect to the kernel without debugging mode of the operating system.

II. WINDBG

The WinDbg debugger is one of the most advanced debuggers for Windows operating systems. WinDbg is claimed by developers, because it extracts symbolic information from applications and libraries, displays the contents of internal Windows data structures, performs remote debugging of a physical or virtual machine. WinDbg can be used to debug user applications, device drivers, the operating system itself in kernel mode, to analyze memory dumps in kernel mode created after the so-called Blue Screen of Death, which occurs when an error is issued. It can also be used to debug custom mode crash dumps. WinDbg supports several debugging modes: debugging the local process, debugging the kernel, and remote debugging.

With local debugging the WinDbg running in the debugged system is easily detected by applications. Remote debugging requires the operation of the OS kernel in debugging mode, when the kernel using the KdSrv service generates various debugging information for responding to requests from a remote debugger, which also reveals system control (Fig. 1).

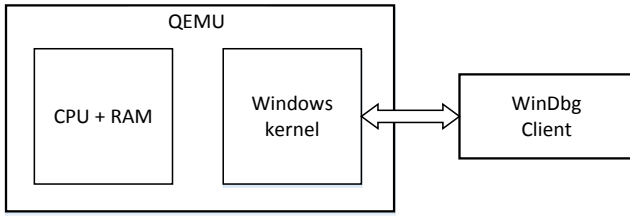


Fig. 1. Directly connecting the WinDbg to the kernel

III. STEALTH DEBUGGING

We developed for the QEMU emulator a mechanism for stealth debugging, which allows WinDbg to be remotely connected. The mechanism is an analysis module built into the emulator, and turns out to be an external tool in relation to the guest system. The needs of the KdSrv service in the core of the system being debugged is not required - the analysis module itself extracts the necessary data from the system and transfers it to the remote debugger (Fig. 2). The programs running in the guest system can not track the presence of the connected debugger through functions such as `IsDebuggerPresent` or through the state of the hardware registers.

One way to remotely debug the kernel using the WinDbg debugger is to debug through the COM port, while the interaction between the computers takes place via a private KDCOM protocol, the specification of which has been restored. One of the computers in this case is represented by a virtual machine, the second is an instrumental computer with Windows OS where this machine is started. Running WinDbg connects to the emulator via a named pipe, through which the COM-port of the virtual machine is forwarded.

The developed module for the emulator fully implements the KDCOM protocol, within the framework of the restored specification, so the debugger WinDbg interacts with it, as with the debugging module of the Windows kernel, without noticing the substitution. It should be noted that the use of the QEMU emulator as a runtime opens the possibility of debugging during deterministic playback of the virtual machine. The recorded scenarios can be debugged many times in the emulator, which would not be possible if the Windows debug module running inside the guest system were used.

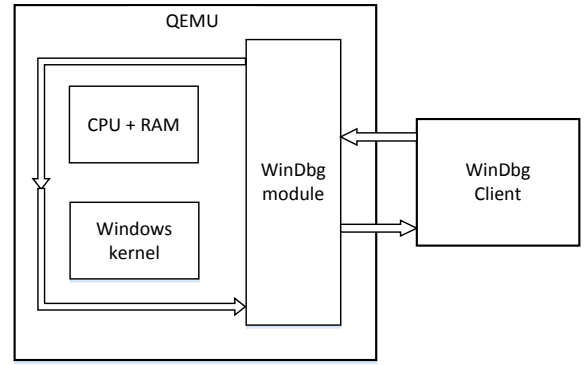


Fig. 2. Connecting the WinDbg to the kernel via the module

IV. RESULTS AND CONTRIBUTIONS

The developed module supports almost all features of WinDbg regarding remote debugging, besides interception of events and exceptions. It is open source project placed in: github.com/ispras/qemu/tree/windbg. The official community recognized the module as useful. In addition, patches have already been prepared for inclusion in the official repository.

ACKNOWLEDGMENT

The work was partially supported by RFBR.

REFERENCES

- [1] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. // Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, 2014, pp. 447-458
- [2] P. Dovgalyuk, Pavel. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, pages 553-556, CSMR '12, Washington, DC, USA, 2012. IEEE Computer Society.
- [3] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A fistful of red-pills: how to automatically generate procedures to detect CPU emulators. In Proceedings of the 3rd USENIX conference on Offensive technologies (WOOT'09). USENIX Association, Berkeley, CA, USA, 2-2.
- [4] IDA Pro Disassembler. [HTML] (<https://www.hex-rays.com/products/ida/index.shtml>).
- [5] Winbagility. [HTML] (<https://winbagility.github.io/>).

An Interactive Specializer Based on Partial Evaluation for a Java Subset

Igor A. Adamovich¹

Ailamazyan Program Systems Institute
Russian Academy of Sciences
4a Peter the First str.
Veskovo, Yaroslavl region, 152021 Russia
i.a.adamovich@gmail.com

Andrei V. Klimov²

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq.
Moscow, 125047 Russia
klimov@keldysh.ru

Abstract—Specialization is a program optimization approach that implies the use of a priori information about values of some variables. Specialization methods are being developed since 1970s (mixed computations, partial evaluation, supercompilation), but unfortunately they have not reached the level needed for their wide application in practice. The task of specialization requires much greater human involvement into the specialization process and the analysis of its results and conducting computer experiments than in the case of common program optimization in compilers. Hence, specializers should be embedded into integrated development environments (IDE) familiar to programmers and appropriate interactive tools should be developed.

In this paper we provide a work-in-progress report on preliminary results of development and implementation of an interactive specializer based on partial evaluation for a subset of the Java programming language. The specializer has been implemented within the popular Eclipse IDE. Scenarios of the human-machine dialogue with the specializer and interactive tools to compose the specialization task and to control the process of specialization are under development.

An example of application of the current version of the specializer is shown. The residual program runs several times faster than the source one.

Index Terms—program analysis, program transformation, interactive program specialization, partial evaluation, object-oriented language, integrated development environment

I. INTRODUCTION

The method of program specialization known as *partial evaluation* was invented more than 30 years ago along with the achievement of the famous result [1], [2] of evaluation of the First, Second and Third Futamura projections [3]–[5] for a tiny List subset. The first round of research was completed in early 1990s when the main textbook on partial evaluation had been published [2]. A lot of programming problems were found to be solved by program specialization (the most known being the generation of a compiler from an interpreter by the Second Futamura Projection) and the emergence of a new class of program development tools based on specialization were

expected. Some other program specialization techniques, e.g., *supercompilation* [6], [7], has been developed in parallel as well. However, it is surprising, that even after three decades, these promising methods have not been put into the wide programming practice. One may wonder: What is the reason?

Our hypothesis is that that main expectation that governed the development of specializers was wrong. The developers of these methods hoped that specializers can work in fully automatic mode and they just needed to invent some finitely many features and improvements that solve the problem, after which “the great goal” would be achieved and happy programmers started using the new tools. They expected that specializers could work in the similar “black-box mode” as optimizing compilers. However this did not happen. The time and space complexity of the program transformations that are necessary for specialization, turned out to be much higher than the complexity of program optimizations that can be used as “black boxes” with short and predictable run time and consumed memory.

We argue that automatic methods of program optimization have reached certain inherent limits. In order to develop and use more powerful tools, we must give up the expectations that the program analysis and transformation systems will operate in automatic mode without human intervention. Program specializers possess too many degrees of freedom and choice, which can not be resolved by the algorithms of their kind and, therefore, should use human help.

Based on this observation, we put forward the goal of construction of an interactive specializer embedded in a habitual integrated development environment (IDE) such as Eclipse [8]. Eclipse provides a rich open-source toolkit referred to as Java development tools (JDT) [9], which allows a developer to deal only with essential tasks of analysis, visualization and transformation of Java code. Adequate human-machine dialogue tools to control the specializer and deal with the results of specialization are to be developed.

We would like to emphasize that there is strict separation of concerns between the machine and the human: the specializer guaranties the functional equivalence of program transformation and the user is responsible to control the specializer in such a way that it produces the code that satisfied his goals

¹Supported by RFBR research project No. 18-37-00454 (contribution: development of interactive methods of partial evaluation, design of the architecture and implementation of the specializer, analysis of related works).

²Supported by RFBR research project No. 16-01-00813 (contribution: problem statement, design of methods based on the existing approaches, supervision, analysis of related works).

```

public class AckermannExample {
    public final long A (long x, long y) {
        if (x == 0) return y + 1;
        else if (y == 0) return A(x - 1, 1);
        else return A(x - 1, A(x, y - 1));
    }

    @Specialize
    public long test(long y) {
        return A(3, y);
    }
}

```

Figure 1. Source code of Ackermann function with BT annotation

and needs (which the machine does not know).

We think that partial evaluation is better suited than other specialization methods (like supercompilation) for human-machine dialogue organized in such a way that the user comprehends what is happening in the specializer, receives valuable and interesting information about his code, is capable of adjusting the source code to be better specialized and controls the specializer. The reason is that the method of partial evaluation consists of two stages:

- *binding-time analysis* (BTA) of source code that selects the parts of the code that are to be evaluated at specialization time, and
- *residual program generation* (RPG) that follows the information supplied by BTA, performs specialization proper and produces the resulting code (referred to as *residual*).

A pleasant feature of BTA is that its result (called *BT annotation*) may be naturally shown on the source code by highlighting and due to such visualization the residual code is intuitively predictable. We hope that this will allow for easy adoption of specializers as new programming tools by rank-and-file programmers.

Terminological remark. In the theory of partial evaluation the parts of source code to be evaluated during specialization are called *static*. The other source code that is transferred to the residual program (*residualized*) is referred to as *dynamic*. The term *static* conflicts with the *static* modifier in Java and the term *dynamic* may be confused with the run-time notions. That is why we avoid using these words in the partial evaluation sense and use abbreviations *S* and *D* instead, e.g., *S*-annotation, *D*-annotation, *S*-code, *D*-code, *S*-part and *D*-part of a program.

The contributions of this paper are as follows:

- We show the first results of development of the Java specializer, where partially evaluated code is restricted to operations on primitive types.
- We demonstrate the work of the specializer by an example of specialization of the Ackermann function with respect to the first argument.
- We discuss some of the details of implementation in Eclipse and the methods and features to be implemented in future.

```

public class AckermannExample {
    public long test(long y) {
        return A_3(y);
    }

    public final long A_3(long y) {
        if (y == 0) return A_2(1);
        else return A_2(A_3(y - 1));
    }

    public final long A_2(long y) {
        if (y == 0) return A_1(1);
        else return A_1(A_2(y - 1));
    }

    public final long A_1(long y) {
        if (y == 0) return A_0(1);
        else return A_0(A_1(y - 1));
    }

    public final long A_0(long y) {
        return y + 1;
    }
}

```

Figure 2. Residual code of Ackermann function

The outline of the paper is as follows. In Section II we present the basics of partial evaluation for Java by the example of specialization of the Ackermann function. In Section III a bird-eye view of the implementation of the specializer in the Eclipse IDE is presented. Section IV contains a survey of related works in comparison with our specializer. In Section V we conclude.

II. JAVA SPECIALIZATION BY EXAMPLE

Figures 1 and 2 contain screenshots of the source and residual code of the Ackermann function made from the running specializer in Eclipse IDE. The method `A` implements the Ackermann function and the method `test` invokes it with the first constant argument 3. The Java annotation `@Specialize` at the method `test` specifies that it should be specialized, i.e., its body is to be replaced with the residual code and the specialized versions of the methods that it invokes are to be generated and added to the program. The names of the methods `A` and `test` in their headers are marked in orange in order to show that they are involved in BTA. The bodies of these methods are analyzed and annotated: green highlighting marks *S*-parts of code. (You see gray highlighting in Figure 1 if you read this paper in a monochrome print).

A. Binding-Time Analysis

The BTA algorithm for variables and operations of primitive types is rather straightforward. First, all constants are annotated with *S*. Then recursively: subexpressions containing only *S*-parts become *S*; local variable declarations and assignments with *S* right-hand sides become *S*; method parameters that

correspond to S arguments at all points of invocation become S ; in case of conflict of several invocations of the same method the conflicting parameter becomes D ; a conflict on several assignments to a local variable turns it to D as well; an `if` statement with the S conditional expression is annotated with S regardless of the annotation of its branches (this means that `if-else` will disappear while one of the branches will be residualized); other control statements are analyzed and annotated similarly. When recursion reaches a fixed point, the remaining parts of code are annotated with D . D -parts are not highlighted in Figure 1.

This mode of operation of BTA, when each code fragment gets univocal annotation S or D , is referred to as *monovariant*. The more general mode when several versions of annotation are allowed is called *polyvariant*. The current version of BTA is monovariant. In future we plan to implement polyvariant BTA for classes and reference types according the theory developed in [10]–[18].

Monovariant BTA on primitive types can be defined formally as abstract interpretation on a lattice with 3 elements: $undefined \leq S \leq D$.

As an illustration of monovariance, notice that method `A` is invoked 3 times in the source code, one of which has both S arguments, another 2 invocations have the first S argument and the second one is D . Nevertheless, the first invocation is processed in the same way as the other two with the second S argument assigned to the D formal parameter.

B. Residual Program Generation

At the generation stage, partial evaluation starts from the method with the `@Specialize` annotation and recursively visits all invoked methods in turn. Notice that, since all statements and methods with side effects are considered D and hence are residualized rather than executed at specialization time, the order of specialization of methods does not matter.

For each of the specialized methods, several residual versions can be produced — one for each combination of values of S arguments. They got different names of the form (in the current version): *source-name_number*. They have only those parameters that correspond to D parameters in the source code.

The current version of the specialized can loop forever if infinitely many values of S arguments are generated. The production version of the specialized should contain special debugging means to gracefully leave such situations. This is our future work.

In Figure 2 there are 4 versions of residual method `A` corresponding to values 0, 1, 2, 3 of its first argument. Notice that because of monovariance the invocations `A_2(1)`, `A_1(1)`, and `A_0(1)` has not being evaluated, since the constant 1 correspond to the D parameter of method `A`.

C. Running Source and Residual Programs

We have chosen this example for presentation, since it demonstrates all main features of the current version of the specialized. We did not expect a significant speed-up as it seemed that asymptotically the number of method invocations

was almost the same and the invocations were the most expensive operations in this example. Thus we were very surprised when the speed-up was about 3 times.

The obtained acceleration can be explained by several reasons. First, calculation showed that the specialized version performs 1.86 times less Java byte code instructions. Second and more important, it is natural to suppose that the JIT compiler in JVM performs inlining of those specialized method that are simpler and more compact than in the source code.

This example illustrates the principle, which we observed many times in experiments with various specializers: A specialized does not replace the classic optimizing compilers. Rather, we observe “composition” of optimizations by a specialized and a low-level optimizing compiler and hence multiplication of speed-ups. Residual code produced by specializers is more amendable for classic optimizations than code written by a human being. We may conclude that specialization opens up additional opportunities for program optimization.

III. ARCHITECTURE OF SPECIALIZER

The specialized has been implemented in the Eclipse development environment (IDE) [8]. The IDE has open source code and provides points and tools to extend it. The basis for Eclipse extension is the concept of a plug-in. Each plug-in is an archive JAR file containing a so-called manifest, a set of files describing the dependencies of the plug-in and the possibility of its extension (extension points). Other plug-ins can add their functionality to these extension points. For example, one might want to add his toolbar extensions to an already implemented toolbar plug-in.

A small tool is usually implemented as a separate plug-in, while a large one is often provided as a set of plug-ins. Our specialized is implemented as three Eclipse plug-ins. The general structure of Eclipse and the dependencies of the specialized (i.e., the plug-ins required for its operation) are shown in Figure 3.

The specialized consists of the following plug-ins:

- A plug-in `SpecCore` is the core of the specialized, which implements its main functionality.
- A plug-in `SpecMarkers` is responsible for text highlighting in the Eclipse editor in accordance with the annotation produced by the `SpecCore` plug-in.
- A plug-in `SpecMenus` implements interactions with various menus (including context menus) and toolbars to provide a user-friendly interface.

The `SpecCore` implements the binding-time analysis (BTA) and the generation of a residual program. When analyzing the source program the plug-in `SpecCore` uses the abstract syntax tree (AST) built by the Eclipse Java development tools (JDT). JDT is a set of plug-ins that provides us with an easy way to manipulate Java source code.

The second of the three plug-ins that form the specialized is the `SpecMarkers` plug-in. It is responsible for highlighting the source code, which allows the programmer to see which parts of the program are evaluated at specialization time and

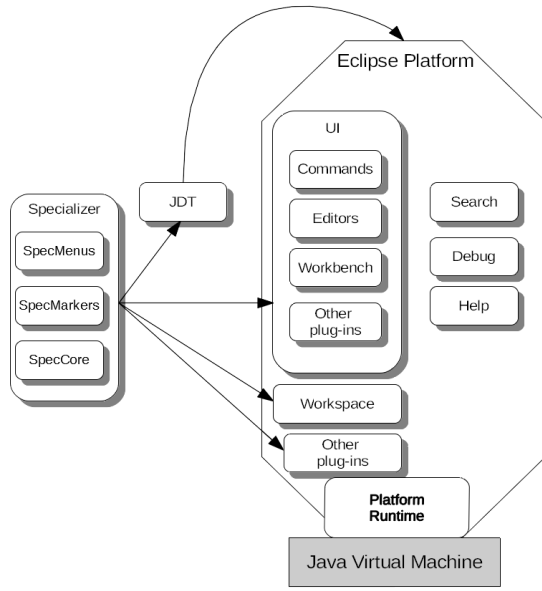


Figure 3. Architecture of Eclipse and the specizer basic dependencies

which are residualized. This helps him to understand how to change the code to provide better specialization.

The last part of the specizer is the SpecMenus plug-in. This plug-in uses the extension points of other plug-ins to add the necessary elements to some menus. It adds two new buttons to the main toolbar of Eclipse: enable/disable the highlighting and the “generate optimized Java files” button. Also this plug-in adds items to the context menu of the Project Explorer and Package Explorer views.

IV. RELATED WORK AND COMPARISON

A lot of works are devoted to partial evaluation for functional languages. The book [2] summarizes the first wave of development of this method.

Later on, research into partial evaluation for imperative “Algol-like” languages [19], [20] and C [21] was performed. In early 1990’s, the first (to our knowledge) specizer for C was developed, called C-MIX [21], [22]. Chapter 11 of the book [2] contains its detailed presentation. C-MIX specializes a program in three stages. The first stage is the analysis of references. For each reference variable, a set of the variables that it can refer to is built. If the analysis finds that several variable references can refer to the same memory, they are labeled identically. The second stage is the construction of a binding-time annotation of the source code. References to the same memory area are annotated identically. In case of conflicts, the annotation is reduced to \perp as usual. The third stage is the generation of the residual program.

Specialization of reference types in Java can be similar to elaboration of pointers in C-MIX. However, Java stricter typing and managed run-time can be leveraged for deeper specialization. The current version of our specizer annotates all reference variables \perp and, therefore, they are left unchanged.

In the future, we plan to add the binding-time analysis of reference types. Unlike C-MIX, we expect that our specizer will continue to work in two stages — without the reference analysis phase.

Further development of ideas of C-MIX led to the creation of a specizer of programs written in C, called Tempo [23], [24]. This specizer is much like C-MIX.

The next important step was the development of the first specizer for an object-oriented language — JSpec for Java [25]. JSpec uses the Harissa compiler [26] to translate the Java program into C. Then the Tempo specizer mentioned above transforms the program. The obtained C-representation of a specialized Java program is mapped back into Java using the Assirah translator [25]. Finally, the AspectJ tool weaves the specialized program with the source program to get the executable Java bytecode. The main limitation of JSpec is that it is capable of partially evaluating only immutable classes and objects, while mutable objects are always residualized. Our goal is to waive this restriction.

The most advanced (to our knowledge) partial evaluation method for object-oriented languages like C# and Java has been developed in CILPE [10]–[18], a partial evaluator for Common Intermediate Language (CIL), the bytecode of the Microsoft .NET Framework. It supports almost all of the basic constructs of object-oriented languages such as C# and Java. In CILPE, a new concept of a binding-time heap (BT heap) has been introduced. The BT heap is an abstract description of the state of the run-time heap, which allows us to separate reference type data into evaluated at specialization time and residualized and to avoid the use of the reference analysis stage as in C-Mix. As a result of specialization, some of the objects are no longer created in the residual program, and if necessary, local variables are used instead of object fields. We will base on the results of this research in our future work to implement BTA of classes and partial evaluation of objects.

A relatively new specizer of Java programs is Civet [27]. Civet is based on a so-called Hybrid Partial Evaluation (HPE) approach. Specialization in HPE is performed in online mode, i.e., in one pass, while the programmer can specify which parts of the program have \perp -annotation. On the contrary, in our specizer we choose the offline approach, i.e., the residual program is built at the stage of generation of the residual program after the completion of the binding-time analysis, where information about the \perp -parts of the program is collected automatically rather than specified by the user as in Civet¹.

PE-KeY [28] is a partial evaluator for Java programs based on the KeY verification system [29]. PE-Key works in two stages. At the first stage, the program is executed in a symbolic form with the application of a special set of rules. At the second stage, a residual program is synthesized, while the rules are applied in the opposite direction. The PE-KeY approach is similar to the classical offline specialization

¹For discussion of the features of and differences between online and offline partial evaluation see [2, Chapter 7].

that our specializer uses: a specialized program is produced in two stages. However, in the first stage of PE-KeY, the program is executed symbolically, while our binding-time analysis performs an abstract interpretation of the program. In addition, due to limitations of the KeY verification system, PE-KeY does not support floating-point arithmetic, while our specializer supports.

JSpec, Civet, PE-Key deal with objects at specialization time, while the current version of our specializer annotates classes and variables of reference types with \mathbb{D} and thus residualizes them unchanged. The extension of our specializer to partial evaluation of classes and objects is our future work.

The specializers considered above interact with the user through the command line, so it's extremely difficult to use them. In order for the specialization to be widely used, it is required to develop the methods of interaction with the user and to embed the specializer into an integrated development environment convenient for the programmer, what we are implementing in our specializer. This is a crucial difference.

We know about just one work on partial evaluation carried out in a practical setting — the GraalVM toolkit in Oracle Labs [30], [31]. The toolkit is designed for defining domain-specific languages by interpreters and, nevertheless, achieving high-performance by using a specializer. The first Futamura projection provides an opportunity for such acceleration (see [3], [4] and [2, Chapter 1.5.1]): given a program and an interpreter that executes the program, GraalVM specializes the interpreter with respect to a part of the given program and produces the machine code of this part. The resulting code is executed much faster than the original one in the interpreter. The main goal of GraalVM is to provide a technology similar to just-in-time (JIT) compilation for the developer of a programming language without the need to implement the complex machinery of JIT. The interpreter specialization in GraalVM is not automatic and uses prompts by the interpreter developer. This case of implementation of partial evaluation confirms that practical application of specialization requires guidance from the programmer. We conduct our research in the same direction: methods and tools are being developed to provide the programmer with information about his program behavior under specialization and levers to control the partial evaluation processes.

V. CONCLUSION

In this paper we put forward the task of development of an interactive specializer. We argue that the current stage of program specialization methods has reached certain limits because the previously implemented specializers do not imply the participation of the user in the process of specialization. Our specializer uses the offline partial evaluation approach, where the program transformation is performed in two stages — binding-time analysis (BTA) and residual program generation (RPG). We briefly described the architecture of our interactive specializer in the Eclipse development environment.

We illustrated the work of the specializer with the famous example of the Ackermann function and the result of its

specialization with respect to its first argument. The specialized program runs several times (about three) faster than the original one.

We see the following directions for further development of the specializer:

- to develop and implement binding-time analysis and residual program generation for classes and objects;
- to implement interactive tools for making a specialization task and controlling the process of binding-time analysis and residual program generation;
- to implement tools to visualize the correspondence of between source and residual code;
- to demonstrate that a well-developed specializer can convert well-structured high-level human-oriented code, which can not be automatically parallelized, into code that can be parallelized by existing methods and tools;
- to prepare demo programs that benefit from specialization, for example, building a compiler from an interpreter;
- to generalize the binding-time analysis from monovariant to polyvariant;
- to develop an interactive tracer (similar to run-time debuggers) that allows the user to observe the analysis and generation processes in order to improve the behavior of his code under specialization.

AVAILABILITY

The current version of our specializer is available at <ftp://ftp.botik.ru/rented/iaadamovich/Specializer/>.

ACKNOWLEDGMENT

We are grateful to our friends and colleagues Yuri Klimov, Arkady Klimov, Sergei Romanenko, Sergei Abramov for valuable advices on specialization methods in general and partial evaluation in particular and constructive feedback on the design of our specializer system.

REFERENCES

- [1] N. D. Jones, P. Sestoft, and H. Søndergaard, "An experiment in partial evaluation: the generation of a compiler generator," in *Rewriting Techniques and Applications*, ser. Lecture Notes in Computer Science, J.-P. Jouannaud, Ed., vol. 202. Springer-Verlag, 1985, pp. 124–140.
- [2] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993. [Online]. Available: <http://www.itu.dk/~sestoft/pebook/pebook.html>
- [3] Y. Futamura, "Partial evaluation of computation process — an approach to a compiler-compiler," *Systems, Computers, Controls*, vol. 2, no. 5, pp. 45–50, 1971.
- [4] —, "Partial evaluation of computation process — an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, no. 4, pp. 381–391, Dec 1999, updated and revised version of [3]. [Online]. Available: <http://doi.org/10.1023/A:1010095604496>
- [5] —, "EL1 Partial Evaluator (Progress Report)," Center for Research in Computing Technology, Harvard University, Tech. Rep., 1973. [Online]. Available: <http://fi.ftrm.info/PE-Museum/EL1.PDF>
- [6] V. F. Turchin, "The concept of a supercompiler," *Transactions on Programming Languages and Systems*, vol. 8, no. 3, pp. 292–325, 1986.
- [7] —, "Supercompilation: techniques and results," in *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25–28, 1996. Proceedings*, ser. Lecture Notes in Computer Science, D. Bjørner, M. Broy, and I. V. Pottosin, Eds., vol. 1181. Springer, 1996, pp. 227–248.

- [8] Eclipse Foundation, "Eclipse integrated development environment (IDE)." [Online]. Available: <https://www.eclipse.org>
- [9] —, "Eclipse Java development tools (JDT)." [Online]. Available: <https://www.eclipse.org/jdt>
- [10] Yu. A. Klimov, "An approach to polyvariant binding time analysis for a stack-based language," in *First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008*. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008, pp. 78–84. [Online]. Available: <http://meta2008.pereslavl.ru/accepted-papers/paper-info-6.html>
- [11] —, "Osobennosti primeneniya metoda chastichny`x vy`chislenij k specializacii programm na ob`ektno-orientirovanny`x yazykax [Program specialization for object-oriented languages by partial evaluation: approaches and problems]," *Preprinty` IPM im. M. V. Keldy`sha [Keldysh Institute Preprints]*, no. 12, 2008, (in Russian). [Online]. Available: <http://library.keldysh.ru/preprint.asp?id=2008-12>
- [12] —, "Vozmozhnosti specializatora CILPE i primery` ego primeneniya k programmam na ob`ektno-orientirovanny`x yazy`kax [Specializer CILPE: examples of object-oriented program specialization]," *Preprinty` IPM im. M. V. Keldy`sha [Keldysh Institute Preprints]*, no. 30, 2008, (in Russian). [Online]. Available: <http://library.keldysh.ru/preprint.asp?id=2008-30>
- [13] —, "SOOL: ob`ektno-orientirovanny`j stekovy`j yazy`k dlya formal'nogo opisaniya i realizacii metodov specializacii programm [SOOL: an object-oriented stacked-based language for specification and implementation of program specialization techniques]," *Preprinty` IPM im. M. V. Keldy`sha [Keldysh Institute Preprints]*, no. 44, 2008, (in Russian). [Online]. Available: <http://library.keldysh.ru/preprint.asp?id=2008-44>
- [14] —, "Specializator CILPE: analiz vremen svyazi`vaniia [Specializer CILPE: binding time analysis]," *Preprinty` IPM im. M. V. Keldy`sha [Keldysh Institute Preprints]*, no. 7, 2009, (in Russian). [Online]. Available: <http://library.keldysh.ru/preprint.asp?id=2009-07>
- [15] —, "Specializator CILPE: generaciya ostatocnoj programmy` [Specializer CILPE: residual program generation]," *Preprinty` IPM im. M. V. Keldy`sha [Keldysh Institute Preprints]*, no. 8, 2009, (in Russian). [Online]. Available: <http://library.keldysh.ru/preprint.asp?id=2009-08>
- [16] —, "Specializator CILPE: dokazatel'stvo korrektnosti [Specializer CILPE: correctness proof]," *Preprinty` IPM im. M. V. Keldy`sha [Keldysh Institute Preprints]*, no. 33, 2009, (in Russian). [Online]. Available: <http://library.keldysh.ru/preprint.asp?id=2009-33>
- [17] —, "Specializaciya programm na ob`ektno-orientirovannyx yazykax metodom chastichnyx vy`chislenij [Specialization of programs in object-oriented languages by partial evaluation]," Ph.D. dissertation, Keldysh Institute of Applied Mathematics of RAS, Moscow, Russia, Nov 2009, (in Russian). [Online]. Available: http://pat.keldysh.ru/~yura/publications/2009.10-Klimov-Disser-Specializaciya_programm_na_ob`ektno-orientirovannyx_yazykah.pdf
- [18] —, "Specializator CILPE: chastichny`e vy`chisleniya dlya ob`ektno-orientirovanny`x yazykov [Specializer CILPE: Partial evaluation for object-oriented languages]," *Programmy`e sistemy`: teoriia i prilozheniia [Program Systems: Theory and Applications]*, no. 3(3), pp. 13–36, 2010, (in Russian). [Online]. Available: http://psta.psiras.ru/read/psta2010_3_13-36.pdf
- [19] M. A. Bulonkov and D. V. Kochetov, "Practical aspects of specialization of Algol-like programs," in *Dagstuhl Seminar on Partial Evaluation*, ser. Lecture Notes in Computer Science, O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110. Springer, 1996, pp. 17–32.
- [20] A. P. Ershov and V. E. Itkin, "Correctness of mixed computation in Algol-like programs," in *MFCs*, ser. Lecture Notes in Computer Science, J. Gruska, Ed., vol. 53. Springer, 1977, pp. 59–77.
- [21] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994, (DIKU report 94/19).
- [22] —, "Binding-time analysis and the taming of C pointers," in *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, D. Schmidt, Ed., 1993.
- [23] C. Consel, J. L. Lawall, and A.-F. L. Meur, "A tour of Tempo: a program specializer for the C language," *Sci. Comput. Program.*, vol. 52, no. 1–3, pp. 341–370, 2004.
- [24] A. L. Meur, J. L. Lawall, and C. Consel, "Towards bridging the gap between programming languages and partial evaluation," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*, Portland, Oregon, USA, January 14–15, 2002, P. Thiemann, Ed. ACM, 2002, pp. 9–18. [Online]. Available: <http://doi.acm.org/10.1145/503032.503033>
- [25] U. P. Schultz, J. L. Lawall, and C. Consel, "Automatic program specialization for Java," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 452–499, 2003.
- [26] G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: A flexible and efficient Java environment mixing bytecode and compiled code," in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies (COOTS)*, June 16–20, 1997, Portland, Oregon, USA, S. Vinoski, Ed. USENIX, 1997, pp. 1–20. [Online]. Available: <http://www.usenix.org/publications/library/proceedings/coots97/muller.html>
- [27] A. Shali and W. R. Cook, "Hybrid partial evaluation," *SIGPLAN Not.*, vol. 46, no. 10, pp. 375–390, Oct. 2011. [Online]. Available: <http://doi.acm.org/10.1145/2076021.2048098>
- [28] R. Ji and R. Bubel, "PE-KeY: A partial evaluator for Java programs," in *Proceedings of the 9th International Conference on Integrated Formal Methods*, ser. IFM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 283–295. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30729-4_20
- [29] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds., *Deductive Software Verification — The KeY Book — From Theory to Practice*, ser. Lecture Notes in Computer Science. Springer, 2016, vol. 10001. [Online]. Available: <https://doi.org/10.1007/978-3-319-49812-6>
- [30] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: ACM, 2013, pp. 187–204. [Online]. Available: <http://doi.acm.org/10.1145/2509578.2509581>
- [31] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," *SIGPLAN Not.*, vol. 52, no. 6, pp. 662–676, Jun. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3140587.3062381>

Static Dependency Analysis for Incremental Validation of Semantically Complex Data

Denis Ilyin

Ivannikov Institute for Systems
Programming of the Russian Academy of
Sciences
Moscow, Russia
denis.ilyin@ispras.ru

Natalya Fokina

Ivannikov Institute for Systems
Programming of the Russian Academy of
Sciences
Moscow, Russia
nfokina@ispras.ru

Vitaly Semenov

Ivannikov Institute for Systems
Programming of the Russian Academy of
Sciences
Moscow, Russia
sem@ispras.ru

Abstract — Modern information systems manipulate data models containing millions of items, and the tendency is to make these models even more complex. One of the most crucial aspects of modern concurrent engineering environments is their reliability. The principles of ACID (atomicity, consistency, isolation, durability) are aimed at providing it, but directly following them leads to serious performance drawbacks on large-scale models, since it is necessary to control the correctness of every performed transaction

In the paper, a method for incremental validation of object-oriented data is presented. Assuming that a submitted transaction is applied to originally consistent data, it is guaranteed that the final data representation is also consistent if only the spot rules are satisfied. To identify data items subject to spot rule validation, a bipartite data-rule dependency graph is formed. To automatically build the dependency graph a static analysis of the model specifications is proposed to apply. In the case of complex object-oriented models defining hundreds and thousands of data types and semantic rules, the static analysis seems to be the only way to realize the incremental validation and to make possible to manage the data in accordance with the ACID principles.

Keywords — information systems, ACID, data consistency management, EXPRESS

I. INTRODUCTION

Management of semantically complex data is one of the challenging problems tightly connected with emerging information systems such as concurrent engineering environments and product data management systems [1-4]. Although transactional guarantees ACID (Atomicity, Consistency, Isolation, and Durability) are widely recognized and recommended for any information system, it is difficult to maintain the consistency and integrity of data driven by complex object-oriented models. Often such models are specified in EXPRESS language being part of the STEP standard on industrial automation systems and integration (ISO 10303). To be unambiguously interpretable by different systems the data must satisfy numerous semantic rules imposed by formal models. Maintaining data consistency and ensuring system interoperability become a serious computational problem. Full semantic validation requires extremely high costs, often exceeding the processing time of individual transactions. Periodic validation is possible, but at a high risk of violating rules and losing actual data.

The paper presents an effective method for incremental validation of object-oriented data. An idea of incremental checks is well-understood and was successfully implemented for the validation of such specific data as UML charts, XML documents, deductive databases [5-7]. Unlike the aforementioned results, the proposed method can be applied to semantically complex data driven by arbitrary object-oriented models.

Assuming that a submitted transaction is applied to originally consistent data, it is guaranteed that the final data representation is also consistent if only the spot rules are satisfied. To identify data items subject to spot rule validation, a bipartite data-rule dependency graph is formed. To automatically build the dependency graph a static analysis of the model specifications is proposed to apply. In the case of large-scale models defining hundreds and thousands of data types and semantic rules, static analysis seems to be the only way to realize the incremental validation and to make possible to effectively manage the data in accordance with the ACID principles.

The structure of the paper is as follows. In section 2, we will shortly overview EXPRESS language with an emphasis on the data types and the rule categories admitted by the language. Formal definitions of model-driven data, rules and transactions are also provided. In section 3, we will present a complete validation routine and then explain how an incremental validation can be arranged using the proposed dependency graph. This is accompanied by an example of the model specification. In conclusion, we summarise benefits of the proposed validation method and outdraw future efforts.

II. PRODUCT DATA AND TRANSACTIONS

A. EXPRESS language

Product data models and, particularly, semantic rules can be specified formally in EXPRESS (ISO 2004) language [8]. This object-oriented modeling language provides a wide range of declarative and imperative constructs to define both data types and constraints imposed upon them. The supported data types can be subdivided into the following groups: simple types (character, string, integer, float, double, boolean, logical, binary), aggregate types (set, multi-set, sequence, array), selects, enumerations, and entity types.

Depending on the definition context, three basic sorts of constraints are distinguished in the modeling language: rules for simple user-defined data types, local rules for object types, and global rules for object type extents. Depending on the evaluation context these imply the following semantic checks:

- attribute type compliance (R_0);
- limited widths of strings and binaries (R_1, R_2);
- size of aggregates (R_3);
- multiplicity of direct and inverse associations in objects (R_4, R_5);
- uniqueness of elements in sets, unique lists and arrays (R_6);
- mandatory attributes in objects (R_7);
- mandatory elements in aggregates excluding sparse arrays (R_8);
- value domains for primitive data types (R_9);
- value domains restricting and interrelating the states of separate attributes within objects (R_{10} or so-called local rules);
- uniqueness of attribute values (optionally, their groups) on object type extents (R_{11} or uniqueness rules);
- value domains restricting and interrelating the states of whole object populations (R_{12} or so-called global rules). Value domains can be specified in a general algebraic form by means of all the variety of imperative constructs available in the language (control statements, functions, procedures, etc.).

Certainly, each product model defines own data types and rules. Therefore, semantic validation methods and tools should be developed in a model-driven paradigm allowing their application for any data whose model is formally specified in EXPRESS language. For a more detailed description refer to the mentioned above standard family which regulates the language.

B. Formalisation of models, data and transactions

An object-oriented data model M can be formally considered as a triple $M = \langle T \cup < \cup R \rangle$, where the types $T = \{C \cup S \cup A \cup \dots\}$ are classes C , simple types S , aggregates A and other constructed structures allowed by EXPRESS. Generalization/specialization relations $<$ are defined among these types. Each class $c \in C$ defines a set of attributes in the form $c.a: C \mapsto T$. The attributes $c.a: C \mapsto C$, $c.a: C \mapsto aggregate(C)$ are single and multiple associations which play role of object references. The rules $R = \{R_0 \cup R_1 \cup R_2 \cup \dots \cup R_{12}\}$ define the value domains of typed data in an algebraic way in accordance with EXPRESS. The rules are subdivided into 12 categories enumerated above. Let us define the key concepts that are used in further consideration.

An object-oriented dataset $x = \{o_1, o_2, \dots\}$ is said to be driven by the model $M\langle T, <, R \rangle$ if all the objects belong to its classes: $\forall o \in x \rightarrow typeof(o) \in C \subset T$.

Let a dataset x is driven by the model $M\langle T, <, R \rangle$. All the objects $\{o^*\} \subset x$ such that $subtypeof(o^*) = c \in C \subset T$ are called an extent of the class c on the dataset x . A query

returning the class extent c on the dataset x is called the extent query and is designated as $Q_{extent}(x, c)$.

Let a dataset x is driven by the model $M\langle T, <, R \rangle$. An object set $\{o^*\} \subset x$, $typeof(o^*) = c^* \in C \subset T$ is said to be interlinked with the objects $\{o\} \subset x$, $typeof(o) = c \in C \subset T$ along the association $c.a$ if $\forall o \in \{o\}, o.a \in \{o^*\}$, $\forall o^* \in \{o^*\} \rightarrow \exists o \in \{o\}: o^* \in o.a$. We will denote that as $\{o\} \xrightarrow{c.a} \{o^*\}$.

Let a dataset x is driven by the model $M\langle T, <, R \rangle$. An object set $\{o^*\} \subset x$, $typeof(o^*) = c^* \in C \subset T$ is said to be interlinked with the objects $\{o\} \subset x$, $typeof(o) = c \in C \subset T$ along the route $\{c.a\}$ if $\exists \{o'\} \subset x, \{o''\} \subset x, \dots$, so that $\{o\} \xrightarrow{c.a} \{o'\} \xrightarrow{c'.a'} \{o''\} \xrightarrow{c''.a''} \dots \rightarrow \{o^*\}$. A query returning the objects $\{o^*\}$ interlinked with a given set $\{o\}$ along the route $\{c.a\}$ is called the route query and is designated as $Q_{route}(x, \{o\}, \{c.a\})$. A query returning the objects $\{o\}$ by a given object set $\{o^*\}$ is called the reverse route query and is designated as $Q_{route}(x, \{o^*\}, rev\{c.a\})$.

The object set $x = \{o_1, o_2, \dots\}$ driven by the model $M\langle T, <, R \rangle$ is called consistent if all the rules being instantiated and evaluated are satisfied on this data set: $\forall r \in R \rightarrow r(x) = true$.

Finally, let us introduce the concept of the delta as a specific representation of transactions. Each delta $\Delta(x', x)$ aggregates the changes happened in the dataset $x' = \{o'_1, o'_2, \dots\}$ compared with its original revision $x = \{o_1, o_2, \dots\}$. It is assumed that both revisions are driven by the same model and the objects have unique identifiers that allows to uniquely map the objects and to compute delta in a formal way as $\Delta(x', x) = delta(x', x)$. The delta can be arranged as bidirectional one and then any of the revisions can be restored by the given other: $x' = apply(x, \Delta)$ and $x = apply(x', \Delta^{-1})$.

The delta is represented as a set of elementary and compound changes $\Delta = \{\delta\}$, where each change can be either the creation of an object, or its deletion or modification designated as $\delta_{new(o)}$, $\delta_{del(o)}$, $\delta_{mod(o)}$ correspondingly. The modification, in turn, is represented as a change in the attributes $\delta_{mod(o)} = \{\delta_{mod(o.a)}\}$ that in the case of aggregates is represented by the operations of insertion, removal and modification of the elements $\delta_{mod(o.a)} = \{\delta_{ins(o.a[])}, \delta_{rem(o.a[])}, \delta_{mod(o.a[])}\}$. In what follows, we assume that each creation operation in the delta representation is complemented by the operations of initializing the attributes that are equivalent to the modification operations. Each deletion operation is supplemented by the operations of resetting the attributes to an undefined state, also representable by the modification operations. Regardless of the way, the delta is structured, only elementary operations are taken into account in the context of the studied validation problems.

III. VALIDATION

A. Complete validation

The complete validation routine is provided below (see Figure 1). In a cycle on all objects their attributes are checked

against the rules of the categories $R_1 \cup R_2 \cup \dots \cup R_9$. The checks are performed individually for each attribute provided that the corresponding rules are imposed on their types. In case of detected violations, the error messages are logged. Rules R_{10} are evaluated for entire objects in the same loop. The second cycle is formed due to the need for checks of uniqueness rules R_{11} . Since these rules are declared inside the class definitions, an additional cycle is arranged on the model classes. The rules are evaluated on the class extents. Finally, the third cycle allows to check global rules R_{12} which are defined directly in the model. Such checks are performed for the corresponding class extents.

```

for each object o ∈ x in dataset
  for each attribute o.a in object
    for each attribute rule ∈ R0 ∪ R1 ∪ R2 ∪ ... ∪ R9
      defined for typeof( o.a )
        check rule(o.a), log if violated
    for each local rule ∈ R10 defined for typeof( o )
      check rule( o ), log if violated
  for each class c ∈ C defined in model
    for each uniqueness rule ∈ R11 defined for class c
      check rule( Q_extent( x, rule.c ) ), log if
violated
  for each global rule ∈ R12 defined in model
    check rule( Q_extent( x, rule.c1 ), Q_extent( x,
rule.c2 ), ... ), log if violated

```

Fig. 1. Complete validation routine.

As mentioned above, complete validation of semantically complex product data is a computationally costly task that can cause performance degradation when processing transactions. Incremental validation makes it possible to reduce the amount of checks to be performed.

B. Incremental validation

The proposed incremental validation method is based on the idea of localizing spot rules that can be affected by a transaction and generating a set of semantic checks that is sufficient to detect all potential violations. For this purpose, the dependency graph is built by a given specification of the data model in EXPRESS language. For brevity, we just explain that this structure represents and omit the details of how it can be formed using static analysis of the specification.

The dependency graph is a bipartite graph whose nodes represent the kinds of transaction operations and the categories of semantic rules both defined by the underlying model. An operation node is connected with the rule nodes by directed edges if only such operations can violate the rules being instantiated for particular data. Usually, the semantics of the operations imply what are the data it is applied to. Sometimes the inspected data are apriori unknown and have to be determined by executing corresponding route queries. Therefore, each edge is formed by the dependency structure σ containing both a rule reference $\sigma.rule$ and an optional query route $\sigma.route$. In some sense, the graph reflects the transaction structure as if it contains all possible kinds of changes and the data organisation as if all data types are present and all rules are potentially suffered to violations. As mentioned above, only elementary operations are involved in the dependency analysis.

Thus, the dependency graph enables to determine spot rules that could be violated for particular data due to the accepted transaction. For example, if the node operation is a modification of the object attribute $c.a$ and a rule $r \in R_0 \cup R_1 \cup R_2 \cup \dots \cup R_9$ is defined for its type, then the node $\delta_{mod(c.a)}$ is connected with the rule node r by a corresponding edge. Having a specific operation of this kind $\delta_{mod(o.a)}$, $typeof(o) = c$ in the delta representation the corresponding check $r(o.a)$ can be produced using the dependency edge.

The method of the dependency graph construction is described in more detail in the next section. Still, here we will point out some of its important features.

If the same attribute $c.a$ participates in an expression of the domain rule $r \in R_{10}$ for the class c , then the operation $\delta_{mod(o.a)}$, $typeof(o) = c$ produces the check $r(o)$ for the object o .

If the attribute $c.a$ participates in the uniqueness rule $r \in R_{11}$ defined for the class c , then another dependency edge must be associated with the operation node. In this case, the corresponding check $r(Q_{extent}(x, c))$ must be performed.

There is a more difficult case when the attribute $c.a$ participates in an expression of the domain rule $r \in R_{10}$ defined for the other class c^* . The attribute $c.a$ is assumed to be accessed by traversing associated objects along the route $\{c^*.a^*\}$ from the objects $o^* \in c^*$. Then the operation $\delta_{mod(o.a)}$, $typeof(o) = c$ induces the checks $r(o^*)$ for all $o^* \in Q_{route}(x, o, rev\{c^*.a^*\})$. To identify and perform such checks the operation node must be connected with the evaluated rule node and a route $\{c^*.a^*\}$ must be prescribed to the edge. The dependency analysis of spot rules $r \in R_{12}$ is carried out in a similar way.

Finally, we note that the operations of creating and deleting objects on the assumptions made above can only violate global rules and only in those cases if the cardinalities of class extents are computed. Considering object references as specific attribute types, it is possible to localize some spot rules more exactly. Differing operations on aggregates also leads to better localization of spot rules. For brevity we omit the details how the spot rules can be localized more carefully and provide an example in the next subsection.

```

for each elementary operation  $\delta(o), \delta(o.a) \in \text{delta}$ 
  {  $\sigma$  } = dependency_graph( kindof(  $\delta$  ) )
  for each dependency  $\sigma \in \{ \sigma \}$ 
    switch kindof(  $\sigma.rule$  )
      case attribute_rule :
        check  $\sigma.rule(o.a)$ , log if violated
      case local_rule :
        {  $o^*$  } = Query_route( x, o, rev(  $\sigma.route$  ) )
        for each  $o^* \in \{ o^* \}$ 
          checkset.put(  $\sigma.rule(o^*)$  )
      case uniqueness_rule :
        checkset.put(  $\sigma$  )
      case global_rule :
        checkset.put(  $\sigma$  )
for each check  $\sigma, \sigma(o) \in \text{checkset}$ 
  switch kindof(  $\sigma.rule$  )
    case local_rule :
      check  $\sigma.rule(o)$ , log if violated
    case uniqueness_rule :

```

```

        check  $\sigma$ .rule( Query_extent( x,  $\sigma$ .rule.c ) ),
log if violated
    case global_rule :
        check  $\sigma$ .rule( Query_extent( x,  $\sigma$ .rule.c1 ),
            Query_extent( x,  $\sigma$ .rule.c2 ),... ), log if
            violated

```

Fig. 2. Incremental validation routine.

The validation routine presented in Figure 2 consists in the sequential traversing of delta operations, determining the nodes of the operation semantics, obtaining associated spot rule nodes, evaluating the rules directly or filling the checklist for the subsequent validation. The checklist is organized as an indexed set of records each of which stores references on the validated rule, query and factual data to perform the corresponding check. The use of the checklist is motivated by the fact that some operations lead to repeated checks of the same rules. Indexing of the checklist allows you to exclude repeated records and, thus, to avoid redundant computations. At the same time, the attribute rule checks are always produced once by the modification operations and, therefore, it is more expedient to execute them immediately, without overloading the checklist.

C. Dependency graph construction

To construct the dependency graph, an abstract syntactic tree for the model is built. According to the retrieved data, for all attribute declarations operation nodes are built. Number and types of these nodes constructed for a single attribute depend on its type. For non-aggregate attributes $c.a$ only node $\delta_{mod}(c.a)$, representing modification of the attribute, is built. For aggregate attributes $c.a[]$ three nodes are created: (1) $\delta_{ins}(c.a[])$ – insertion of a new element; (2) $\delta_{mod}(c.a[])$ – modification of an element of the aggregate; (3) $\delta_{rem}(c.a[])$ – removal of an element.

Construction of the dependency graph proceeds with generating rule nodes. We handle construction of nodes for rules R_1 - R_9 and R_{10} - R_{12} differently.

For rules R_1 - R_9 we take all explicit attributes and build rule nodes for each of them. The types of rule nodes depend on the type of the attribute in question. For instance, if it is a bounded string $c.S$, we generate a $R_1(c.S)$ (R_1 – limited width of strings), connected with the node corresponding to the modification of S $\delta_{mod}(c.S)$. Similarly, if an attribute is a bounded aggregate, we construct a node of type R_4 and connect it with the insertion $\delta_{ins}(c.a[])$ and/or removal $\delta_{rem}(c.a[])$ operation nodes of the attribute, depending on the side from which the aggregate is bounded – if it is bounded above, then only with insertion node, if below – with removal, if from both sides – with both of them.

The way of construction of rule nodes for R_{10} - R_{12} is uniform. We start with locating all local rules for R_{10} , all uniqueness rules for R_{11} and all global rules for R_{12} . For each of the rules, we find all attributes used in it. If an attribute is explicit, we only connect its modification with the rule node, and also with insertion and removal, if it is an aggregate used inside a SIZEOF operation. If an attribute is derived, we take its definition and find the attributes used in it; if inverse – we proceed with analyzing the attribute it references. For derived

and explicit attributes, the analysis is performed recursively, until all the explicit attributes, directly and indirectly referenced by them, are located. Then all of them are connected with the rule node corresponding to the rule in question. If during the analysis we find a node that is a function call, we substitute its formal parameters with actual and thus locate the attributes which are used in it; the analysis of a function body with the parameters substituted is completely identical to the analysis of a rule.

An example illustrating the constructed graph is given in the next section.

D. Example of a dependency graph

Let us consider a fragment of the EXPRESS specification of a project management system. Three classes depicted in Figure 3 – *Task*, *Link* and *Calendar* – are its core entities. The meaning of *Task* is self-evident; *Link* represents a connection defining a relation and execution order between two tasks. The fact that between two tasks might be only a single link of one type is reflected in uniqueness rule *ur1*. A *Calendar* defines a typical working pattern: working days, working times, holidays. The calendar can be assigned to specific tasks, and one calendar can be set as a default project calendar, that means that it will be used for tasks for which no task calendar is set. Besides that, it is possible to use an *Elapsed* calendar for a task implying that work will be performed 24/7. Global rule *SingleProjectCalendar* restricts the possible number of project calendars to no more than one. Moreover, local rule *wr3* is used to check that if a task has got a task calendar, it the reference to it must be non-null. One more local rule, *wr2*, restricts the length of an *EntityName* to be between 1 and 32 characters.

```

TYPE LinkEnum = ENUMERATION OF
    (START_START, START_FINISH, FINISH_START,
    FINISH_FINISH);
END_TYPE;

TYPE CalendarRuleEnum = ENUMERATION OF
    (TASK, PROJECT, ELAPSED);
END_TYPE;

FUNCTION TaskIsCyclic (T1 : Task, T2 : Task) :
    BOOLEAN;
    IF (SIZEOF(T1.Parent) = 0) THEN RETURN(FALSE);
    ELSE
        IF ((TaskIsCyclic(T1.Parent[1], T2) = TRUE)
        OR (T1 = T2))
            THEN RETURN(TRUE);
        END_IF;
    END_IF;
END_FUNCTION;

RULE SingleProjectCalendar FOR (Calendar);
WHERE
    wr1: SIZEOF(QUERY(Temp <* Calendar |
    Temp.isProjectCalendar = TRUE)) <= 1;
END_RULE;

TYPE EntityName = STRING;
WHERE
    wr2: (1 <= SELF) AND (SELF <= 32);
END_TYPE;

ENTITY Task;
    ID : INTEGER;

```



```

Name : EntityName;
TaskCalendar : Calendar;
CalendarRule : CalendarRuleEnum;
Children : LIST [0:?] OF Task;
DERIVE
  TaskDuration : Duration := ?;
INVERSE
  Parent : SET [0:1] OF Task FOR Children;
  DownstreamLinks : SET [0:?] OF Link FOR
Predecessor;
  UpstreamLinks : SET [0:?] OF Link FOR Successor;
WHERE
  wr3 : CalendarRule <> CalendarRuleEnum.TASK OR
EXISTS(TaskCalendar);
  wr4 : (SIZEOF(Parent) = 0) OR
(TaskIsCyclic(Parent[1], SELF) = FALSE);
UNIQUE
  ur1 : ID;
END_ENTITY;

ENTITY Link;
  ID : INTEGER;
  LinkType : LinkEnum;
  Predecessor : Task;
  Successor : Task;
UNIQUE
  ur2 : LinkType AND Predecessor.ID AND
Successor.ID;
  ur3 : ID;
END_ENTITY;

ENTITY Calendar;
  ID : INTEGER;
  Name : OPTIONAL EntityName;
  IsProjectCalendar : BOOLEAN;
UNIQUE
  ur4 : ID;
END_ENTITY;

```

Fig. 3. An example of the model specification in EXPRESS language

The dependency graph for this fragment of the specification is shown in Figure 4.

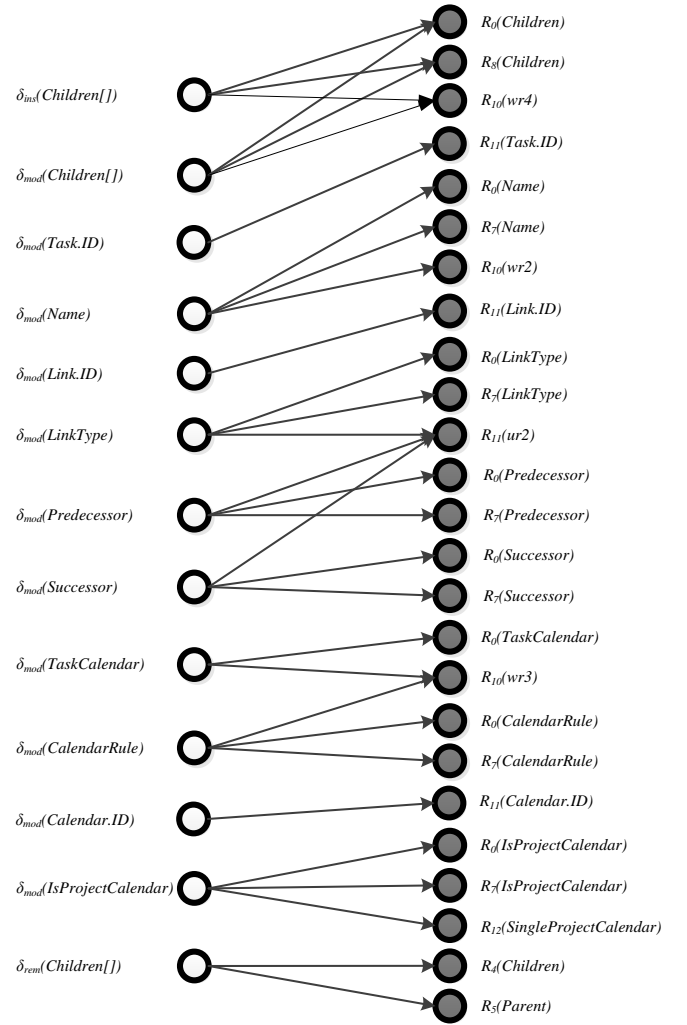


Fig. 4. A fragment of the model dependency graph

Each operation of attribute modification except for removal of elements from the list of task children is connected with the rules validating corresponding attribute type compliance R_0 and availability of defined values for mandatory attributes R_7 . To avoid placement of null values to the list of mandatory elements the rule R_8 should be validated as well after the operations have been performed. The insertion cannot violate multiplicity of the direct and inverse associations as their upper borders are unlimited, but checks R_4 , R_5 should be performed when an element is removed from *Children*. So, the corresponding operation nodes should be connected with the aforementioned nodes of the rules that the operations may potentially violate. As the expression for the local rule *wr3* includes the attributes *CalendarRule* and *TaskCalendar*, the nodes corresponding to the operations of modification of these attributes are connected with the *wr3* rule node. For the rule *wr2* defining the value range of the *EntityName* type, there is a connection between the *EntityName* modification node and the *wr2* rule node. The corresponding edges are assigned by the routes by traversing of which the attributes could be accessed. The expression of

the global rule *SingleProjectCalendar* references only one attribute *IsProjectCalendar*, so the appropriate graph nodes are connected by the edge as well. Modification of any attribute of the *Link* class can affect its uniqueness defined by *ur2*; hence the connections between *LinkType*, *Predecessor* and *Successor* and the uniqueness rule node.

It is also possible that a change affects a constraint not directly but through an inverse association, or even a chain of them, where other classes can be involved. In this case, rules for all the chain of affected classes is added to the checkset. Furthermore, they can be affected not only by direct associations but also by the inverse. For instance, cardinality constraints on inverse aggregate attributes causes insertion of additional rule nodes to the graph.

IV. CONCLUSION

This paper presents the incremental method of model data validation. The method is applicable for semantically complex data driven by arbitrary object-oriented models. It allows to increase the performance of semantic validation and to effectively manage the data in accordance with the ACID principles.

The planned work concerns basically the implementation of the method proposed and its evaluation for industry meaningful product data. The expected positive results will allow its wide introduction into new software engineering technologies and emerging information systems.

REFERENCES

- [1] V.A. Semenov, Product Data Management with Solid Transactional Guarantees, In C.-H. Chen, A.C. Trappey, M. Peruzzini, J. Stjepandić, N. Wognum (eds.): Transdisciplinary Engineering: A Paradigm Shift Series Advances in Transdisciplinary Engineering, IOS Press, 2017, pp. 592-599.
- [2] L. Lämmer and M. Theiss, Product Lifecycle Management, In J. Stjepandić, N. Wognum, W.J.C. Verhagen (eds.): Concurrent Engineering in the 21st Century — Foundations, Developments and Challenges, Springer, 2015, pp. 455-490.
- [3] J. Osborn, Survey of concurrent engineering environments and the application of best practices towards the development of a multiple industry, multiple domain environment, Clemson University, 2009, Accessed: 29/01/2018. Available: http://tigerprints.clemson.edu/all_theses/635/
- [4] M. Philpotts, An introduction to the concepts, benefits and terminology of product data management, Industrial Management & Data Systems, MCB University Press, vol. 96, no. 4, 1996, pp. 11–17.
- [5] X. Blanc, A. Mougenot, I. Mounier, T. Mens, Incremental Detection of Model Inconsistencies based on Model Operations, In: van Eck P., Gordijn J., Wieringa R. (eds): Advanced Information Systems Engineering, CAiSE 2009, LNCS, vol. 5565, Springer, 2009, pp. 32-46.
- [6] C. Xu, C.S. Cheung, W.K. Chan, Incremental Consistency Checking for Pervasive Context, In Proc. the 28th International Conference on Software Engineering, 2006, pp. 292-301.
- [7] J. Harrison, S.W. Dietrich, Towards an Incremental Condition Evaluation Strategy for Active Deductive Databases, In: B. Srinivasan, J. Zeleznikow (eds.): Research and Practical Issues in Databases, World Scientific, 1992, pp. 81-95.
- [8] ISO 10303-11: 2004. Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual, ISO, 2004.

Source Code Augmentation for Supervised Learning

Valeriy Savchenko

Ivannikov Institute for System Programming
of the Russian Academy of Sciences
25 Alexander Solzhenitsyn street
109004, Moscow, Russian Federation
Email: vsavchenko@ispras.ru

Alexander Volkov

Lomonosov Moscow State University
GSP-1 Leninskie Gory
119991, Moscow, Russian Federation
Email: volkovas_174@icloud.com

Abstract—Modern machine learning algorithms rely on large quantities of labeled data. Google’s Open Images, for example, consists of 9 million images annotated with over 6000 labels. However, the process of labeling is expensive because it requires significant manual work. When it comes to source code, costs are even higher in comparison to image or text, considering the level of expertise needed to tag it correctly.

One known method of reducing the manual workload is data augmentation, an automatic way to expand the existing labeled dataset. It is widely used in text and image processing. Augmentation expands the dataset by duplicating each element in the sample, and then performing simple transformations to the copies that preserve the label. For example, an image of a cat may be augmented by rotation or cropping, provided that it remains recognizable as a cat. However, source code augmentation is difficult to implement due to the fact that manipulating the text easily changes the meaning of the original code.

The aim of this study is to develop an instrument that is able to maintain the semantics of the transformed code. In order to achieve this, we use compiler dataflow analyses. Our experiments show that the proposed techniques can increase the size of the sample by orders of magnitude.

Index Terms—data augmentation, code transformation, llvm, compiler, machine learning

I. INTRODUCTION

Software systems grow at an incredible speed; big corporations operate with codebases of millions and tens of millions lines of code. In 2017 alone, Github hosted 1 billion commits[1]. These extensive amounts of data provide a great opportunity to improve software development using machine learning and data analysis. Full or partial automation of everyday activities like bug detection, code review, and program comprehension can significantly improve developer productivity and the quality of the final product.

However, for supervised learning algorithms, one should provide not only data, but also labels. Popular public datasets ImageNet[2] and Google’s Open Images[3] contain millions of annotated images. Manual labeling at that scale is expensive in terms of time and human-power. The cost grows with the amount of work and expertise required to analyze and label each example properly. The cost is heightened further in the case of source code, which will be examined later.

Data augmentation is a known way to expand an existing labeled dataset. It provides new examples by duplicating already labeled ones and modifying them. The benefit is that new examples automatically have labels without any manual

work. The main requirement is that augmentation preserves the original label. In text analysis, for example, if we were to train a model to predict sentiment in tweets, we could expand the dataset through synonym replacements, provided that the meaning of the tweet remained the same.

Label retention becomes a more serious issue in source code augmentation. Unlike image or text augmentations, that generally perform simple heuristic transformations, source code must be altered with more caution. Any non-trivial modification of the program source text can change its behavior. This paper investigates the possibility of using code analysis to perform code augmentation that maintains the meaning of the original program.

Compiler optimization techniques provide us with the necessary framework to achieve this goal. Code transformations that maintain code semantics are known as *safe* or *conservative*[4]. In this paper, we propose three conservative augmentation techniques that help alter instruction order, control flow and call graphs of the modified program. This diversification ensures that features of different natures vary from one augmented example to the next.

This paper is organized into six sections as follows. The next section gives a brief overview of our infrastructure. We propose three new source code augmentation algorithms in Section III. The fourth section examines the generative ability of the developed solution. Section V gives a review of the related work, and our conclusions are drawn in the final section.

II. INFRASTRUCTURE

Our implementation is based on LLVM[5]. LLVM is a modern modular compiler infrastructure. It provides high-level interfaces to perform analyses and transformations over its own intermediate representation (IR). Compilers for many languages use LLVM as their backend, most notably C/C++¹, Rust², Swift³, and Kotlin⁴. LLVM is easily extended, making it a playground for realizing new ideas. It is also can be embedded into other programs (i.e. used as a library), which is rare for such a large project.

¹<https://clang.llvm.org/>

²<https://www.rust-lang.org/>

³<https://www.apple.com/swift/>

⁴<https://kotlinlang.org/>

By using LLVM, we impose the following limitations:

- 1) Augmentation requires the ability to compile the code first, which in turn depends on flags used during compilation such as include directories and macro definitions.
- 2) The features used for machine learning must be extracted from IR and not from the source code. This demands that data scientists have a comprehensive understanding of source code analysis and compiler techniques.

III. TRANSFORMATIONS

A. Instruction shuffling

One of the first ideas for code augmentation is to shuffle instructions, however like English text, this requires a thorough analysis to maintain its meaning. The meaning of a sentence, for example, can be easily changed by swapping object and subject of the sentence. Thus to accomplish conservative instruction shuffling, dependencies should be taken into account. Dependencies can be of two kinds: data and control[6]. In this study, we consider two ways to perform instruction shuffling: within the borders of a single basic block, and within the borders of one function.

1) *Single basic block*: Due to a linear structure of a basic block, we can discard control dependencies. Data dependency in turn can be divided into two categories: pure use-def dependency[7] and aliasing. Since our target languages are C/C++, which allow their users to work with memory directly, not all alias analysis techniques can be used[8]. We selected Steensgaard alias analysis[9] due to its good balance between accuracy and complexity. It is significant considering the fact that precise flow-insensitive alias analysis is \mathcal{NP} -hard[10]. Alternatively use-def chains require no additional computations since they are incorporated into LLVM’s SSA form[11].

Algorithm 1 Instruction shuffling (basic block)

```

1: procedure SHUFFLE( $BB$ )
2:    $N := |BB|$ 
3:    $decided := \emptyset$ 
4:   for  $i := 0$  to  $N - 1$  do
5:      $p := rand(N)$ 
6:      $d := pick^5(\{1, -1\})$  ▷ randomly pick direction
7:     while  $p + d \notin decided \wedge BB_{p+d} \text{ depends}^6 \text{ on } BB_p$  do
8:        $swap(BB_{p+d}, BB_p)$ 
9:        $p := p + d$ 
10:    end while
11:     $decided := decided \cup \{p\}$  ▷ mark instruction as decided
12:  end for
```

The resulting algorithm can be summarized as follows: we randomly select one of the instructions in the basic block and the direction in which we want to move it, then we move it the furthest we can do it while maintaining all of the instruction’s dependencies. Algorithm 1 shows it in more details.

2) *Single function*: In order to shuffle instructions within the borders of a function, we need to tackle the problem of control dependency as well. To simplify our algorithm, we move only instructions that dominate[12] all of their

dependent instructions. In fact, this restriction provides us with a useful property: each basic block of the function that is dominated by the selected instruction, and dominates all dependent instructions of the selected instruction, can be used as a new basic block for this instruction.

The entire procedure is illustrated in Algorithm 2.

Algorithm 2 Instruction shuffling (function)

```

1: procedure MOVE( $Objective$ )
2:    $Source = Objective.parent$  ▷ original basic block
3:    $deps := \{inst.parent : \forall inst \text{ dependent on } Objective\}$ 
4:   if  $\exists BB \in deps \rightarrow Source \neg \text{dominates } BB$  then
5:     return
6:   end if
7:    $D := nearest\ common\ dominator(deps)$ 
8:    $candidates := \emptyset$ 
9:   for  $BB \in DomTree.Descendants(Source)$  do
10:    if  $BB \text{ dominates } D$  then
11:       $candidates := candidates \cup \{BB\}$ 
12:    end if
13:  end for
14:   $Dest := pick(candidates)$ 
15:   $Dest := \{Objective\} \cup Dest$  ▷ place it at the top of Dest
16:   $Source := Source \setminus \{Objective\}$  ▷ remove from Source
```

B. Conversion of a preconditional loop into a postconditional loop

In typical C/C++ code, preconditional loops (`for` and `while`) dominate over postconditional (`do-while`). To artificially increase the presence of the last kind in a sample, we suggest converting pre- into postconditions. Because they are not semantically equal, we need to make additional transformations to keep the original meaning. the only distinction between the two kinds, apart from syntactic differences, is the possibility of the first iteration. Therefore, in order to make it non-mandatory, we duplicate the loop condition and place it before the loop body (as illustrated in Fig. 2).

C. Function inlining and outlining

In order to diversify source code on a higher interprocedural level, we use inlining[13] and outlining[14]. These changes the CFG of each modified function, as well as the call graph of an augmented program.

Our approach is to randomly select function calls to inline and CFG subgraphs to outline. However, not every CFG subgraph can be extracted into a separate function because each function should start with a single entry block that dominates all other basic blocks in the function. Furthermore, to simplify the extraction itself, we limit the resulting function to one exit block as well. Algorithm 3 illustrates the entire procedure. It is noteworthy that for any selected basic block, there is at least one subgraph that satisfies all domination and post-domination conditions. Because every basic block dominates and post-dominates itself.

The juxtaposition of inlining and outlining performed in a random order corresponds to partial inlining; therefore, we always inline a whole function and not part of it.

⁵randomly pick an element from the given set

⁶according to use-def and alias information

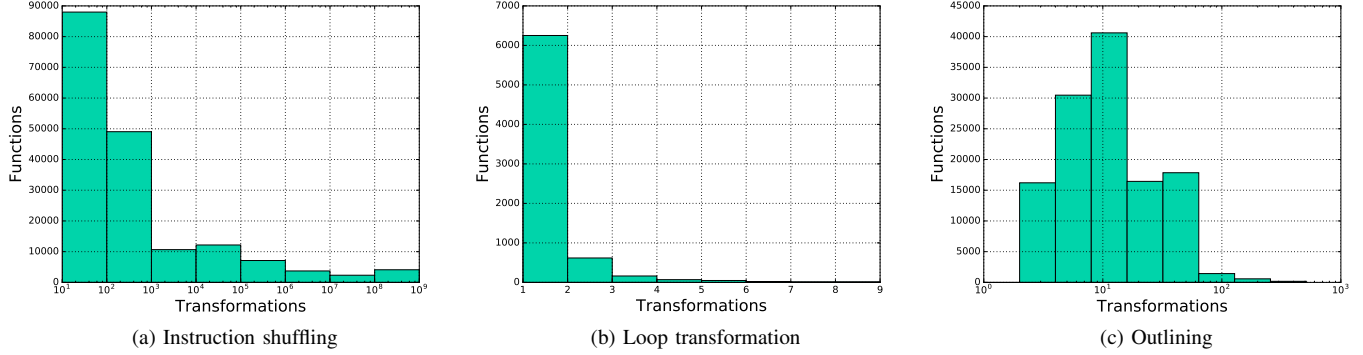


Figure 1: Evaluation results

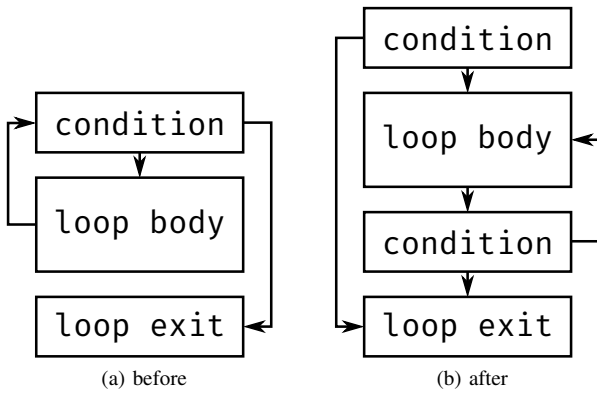


Figure 2: Loop transformation

Algorithm 3 Function outlining

```

1: procedure OUTLINE(Function)
2:   Entry := pick(Function.Blocks)
3:   dominated := DomTree.Descendants(Entry)
4:   post-dominators :=  $\emptyset$ 
5:   for BB  $\in$  dominated do
6:     if BB post-dominate Entry then
7:       post-dominators := post-dominators  $\cup$  {BB}
8:     end if
9:   end for
10:  Exit := pick(post-dominators)
11:  extract code region(Entry, Exit)

```

IV. EVALUATION

Our goal is to research and evaluate the performance of implemented source code transformations. We measure this in terms of the number of possible transformations. Since the original code can vary in size and structure, we use a real-world open-source project (LLVM). Functions are the unit of source code transformation. For each function in the project, we calculate the number of ways this function can be augmented.

Fig. 1 shows the evaluation results. Each of the histograms represents a distribution of functions over a number of possible augmentations. As expected, the most efficient transformation

is instruction shuffling. 50% of all functions in the test project can be augmented in more than 100 ways. On average across all function, the number of transformations reaches 10^6 .

Loop transformations can be applied only to functions with preconditional loops. For LLVM, they make up 2% of all functions. Each of these functions can be transformed in at least one way, 14% of the functions in at least two ways.

Outlining on average produces 22 augmentations per function, and 60% of all functions can be outlined in 10 or more different ways.

Our findings show that even for the big variety in the original dataset, our method is able to enlarge the dataset by the orders of magnitude and augment a significant amount of samples.

V. RELATED WORK

Although source code augmentation is new, it is heavily influenced by adjacent areas of study in computer science. Augmentation in general is widely used for different types of data. The most famous data augmentation techniques come from image processing. Simple transformations like flipping, cropping, and blurring can be easily implemented and provide good results. Even applied to classic problems like MNIST⁷, augmentation can improve existing state-of-the-art models[15], [16], [17]. New examples can even be composed of reference images to create a dataset from scratch[18]. In speech recognition[19], music[20], and sound processing[21], [22], additional noise and distortions expand the scope of the dataset, improving the quality of the trained model. Interesting results were obtained for natural language processing as well, where augmentation can be used on different levels — from generating new words with morphemes[23] to generating questions for visual question answering (VQA)[24]. For some problems, like low-resource language processing[25], data augmentation is the only viable solution to enlarge limited data.

Code augmentation is possible only within strict borders of conservative transformations. It is a vast area of research

⁷<http://yann.lecun.com/exdb/mnist/>

that takes its roots from compiler optimizations. It modifies programs to improve performance, size[26], and energy consumption[27]. All of this makes sense only if the result is functionally equivalent to the original. Compiler optimization is an actively developing domain[28], [29], [30] with a well-formed foundation [4], [31], [32], [33].

Code obfuscation is another similar area of study. Its goal is to deliberately make the original code hard for human understanding. In addition optimization and augmentation, obfuscation also has to maintain code semantics. Obfuscation algorithms include dead, irrelevant, and unreachable code insertion, loop unrolling[34], opaque predicates[35], [36], and CFG dispatching[37]. Even though obfuscation and augmentation pursue different goals, many of the obfuscation methods can be used for augmentation.

VI. CONCLUSION AND FUTURE WORK

Achieving correct source code augmentation requires deep analysis of control and data dependencies. In this paper, we suggested conservative transformation methods. Our experiments demonstrate that our methods are able to increase the size of the dataset by orders of magnitude. This finding lays the foundation for the development of new methods of source code augmentation.

There are two limitations to this approach. First, presented code augmentation requires a compiled code, which means that in order to use code augmentation, compilation flags must be provided for each source file. Next, data scientists are restricted to extracting features from intermediate representations, not the source code itself.

Our next step is to test this approach on real-world deep learning models and evaluate its influence on performance.

REFERENCES

- [1] "The state of the octoverse 2017." <https://octoverse.github.com/>, 2017.
- [2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.
- [3] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy, "Openimages: A public dataset for large-scale multi-label and multi-class image classification," *Dataset available from <https://github.com/openimages>*, 2017.
- [4] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- [5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.
- [6] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, pp. 319–349, July 1987.
- [7] J. C. Reynolds, "Automatic computation of data set definitions," IFIP Congress, 1967.
- [8] V. Raman, "Pointer analysis – a survey," 2004.
- [9] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 32–41, ACM, 1996.
- [10] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Trans. Program. Lang. Syst.*, vol. 19, pp. 1–6, 1997.
- [11] C. Lattner and V. S. Adve, "The llvm instruction set and compilation strategy," 2002.
- [12] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, 1979.
- [13] R. Allen and S. Johnson, "Compiling c for vectorization, parallelization, and inline expansion," *SIGPLAN Not.*, vol. 23, pp. 241–249, June 1988.
- [14] P. Zhao and J. N. Amaral, "Function outlining and partial inlining," *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, pp. 101–108, 2005.
- [15] K. R. Konda, X. Bouthillier, R. Memisevic, and P. Vincent, "Dropout as data augmentation," *CoRR*, vol. abs/1506.08700, 2015.
- [16] Z. Zhong, L. Zheng, G. Kang, S. Li, and Y. Yang, "Random erasing data augmentation," *CoRR*, vol. abs/1708.04896, 2017.
- [17] A. Fawzi, H. Samulowitz, D. S. Turaga, and P. Frossard, "Adaptive data augmentation for image classification," *2016 IEEE International Conference on Image Processing (ICIP)*, pp. 3688–3692, 2016.
- [18] B. Sapp, A. Saxena, and A. Y. Ng, "A fast data collection and augmentation procedure for object recognition," in *AAAI*, 2008.
- [19] T. Ko, V. Peddinti, D. Povey, and S. Khudanpur, "Audio augmentation for speech recognition," in *INTERSPEECH*, 2015.
- [20] B. McFee, E. J. Humphrey, and J. P. Bello, "A software framework for musical data augmentation," in *ISMIR*, 2015.
- [21] B. McFee, E. J. Humphrey, and J. P. Bello, "A software framework for musical data augmentation," in *ISMIR*, 2015.
- [22] X. Cui, V. Goel, and B. Kingsbury, "Data augmentation for deep neural network acoustic modeling," *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5582–5586, 2014.
- [23] M. Silfverberg, A. Wiemerslage, L. Liu, and L. J. Mao, "Data augmentation for morphological reinflection," in *CoNLL Shared Task*, 2017.
- [24] K. Kafle, M. A. Youseffhussien, and C. Kanan, "Data augmentation for visual question answering," in *INLG*, 2017.
- [25] M. Fadaee, A. Bisazza, and C. Monz, "Data augmentation for low-resource neural machine translation," in *ACL*, 2017.
- [26] R. Leupers, *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.
- [27] G. E. Chen, F. Li, M. T. Kandemir, and M. J. Irwin, "Reducing noc energy consumption through compiler-directed channel voltage scaling," in *PLDI*, 2006.
- [28] G. Duboscq, T. W252rthinger, L. Stadler, C. Wimmer, D. Simon, and H. M246ssenb246ck, "An intermediate representation for speculative optimizations in a dynamic compiler," in *VML '13*, 2013.
- [29] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*, 2013.
- [30] K. Ishizaki, A. Hayashi, G. Koblenz, and V. Sarkar, "Compiling and optimizing java 8 programs for gpu execution," *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 419–431, 2015.
- [31] G. Wood, "Global optimization of microprograms through modular control constructs," in *ACM SIGMICRO Newsletter*, vol. 10, pp. 1–6, IEEE Press, 1979.
- [32] M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "A technique of global optimization of microprograms," in *MICRO*, 1978.
- [33] J. D. Ullman, "Fast algorithms for the elimination of common subexpressions," *Acta Informatica*, vol. 2, no. 3, pp. 191–213, 1973.
- [34] C. Collberg, C. Thomborson, D. Low, and D. Low, "A taxonomy of obfuscating transformations. department of computer sciences, the university of auckland," tech. rep., Technical Report 148, July, 1997.
- [35] S. Chow, Y. Gu, H. Johnson, and V. A. Zakharov, "An approach to the obfuscation of control-flow of sequential computer programs," in *International Conference on Information Security*, pp. 144–155, Springer, 2001.
- [36] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 184–196, ACM, 1998.
- [37] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pp. 193–202, IEEE, 2001.

Buffer Overflow Detection via Static Analysis: Expectations vs. Reality

Irina Dudina

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

Lomonosov Moscow State University

GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation

Email: eupharina@ispras.ru

Abstract—Over the last few decades buffer overflow remains one of the main sources of program errors and vulnerabilities. Among other solutions several static analysis techniques were developed to mitigate such program defects. We analyzed different approaches and tools that address this issue to discern common practices and types of detected errors. Also we explored some popular sets of synthetic tests (Juliet Test Suite, Toyota ITC benchmark) and set of buggy code snippets extracted from real applications to define types of defects that a static analyzer is expected to uncover. Our goal is to use this knowledge to enhance our own buffer overrun detector. Now it can perform interprocedural context- and path-sensitive analysis to detect buffer overflow mainly for static and stack objects with approximately 65% true positive ratio. We think that promising directions are improving string manipulations handling and combining taint analysis with our approaches.

I. INTRODUCTION

Buffer overflow is a type of program defect caused by buffer access with index that exceeds buffer's bounds. This can lead to a program crash or even to a security vulnerability. Defects of such kind are still really common, despite all efforts made to eliminate them. There are several techniques one can apply to detect buffer overflows. One approach is to employ testing and dynamic analysis. These methods don't suffer from false positives, but in most cases it's impossible to check all execution paths, so some defects can remain undetected. Another approach is to analyze program code without executing it. In this way one can find a defect on any path, even rarely executed. In this paper we will focus on the latter approach known as static analysis.

We are interested in building a buffer overflow detector that is applicable to large C/C++ programs with millions of lines of code while producing decent analysis performance and quality. Basic properties of the algorithms constituting such a detector are well-known and include among others interprocedural analysis, path sensitivity, and loop handling. However, after initial support for these features has been made and the quality goals achieved, it is unclear which direction to choose for the further improvement. The usual development pace that comes from the customer feedback and own code analysis may be not enough. In the following chapters we'll overview possible sources of inspiration for the buffer overflow detector development, present our short survey that is based on the

buffer overflow-related vulnerabilities sample from the CVE database, then briefly describe our experience of developing an overrun detector as a part of the Svace tool, and present our conclusions from tools and vulnerabilities analysis.

II. BUFFER OVERFLOW DETECTION TECHNIQUES AND TOOLS

There exist many static analysis tools that can detect buffer overflows. In this section we conduct a brief survey on the most popular methods.

Some buffer overflows can be detected during the process of lexical analysis, like in the ITS4 tool [1]. Most common errors and bad patterns can be found at this level. This technique can work really fast and, as it doesn't involve compilation, can be easily applied to any code, even if it is not complete. As a result, such analysis can be performed "on-the-fly" during the process of code development with IDE, so that erroneous patterns are eliminated on the very early coding stage. Of course, such a lightweight method is far from being sound, i.e. it misses many defects. Even changing the name of a variable can prevent such tools from detecting a defect.

To detect more defects a deeper analysis of code is needed. To achieve this many tools use the idea of abstract interpretation [2]. Some tools chose different numerical abstract domains to implement the analysis of integer index values, buffer sizes, and string lengths. These domains include intervals, zones, octagons, affine equalities, interval linear equalities, convex polyhedra, tropical polyhedra, etc. [3]. Tools based on this approaches derive sound relationship between integer values listed above in varying degrees of precision. Soundness is a major advantage of such tools, but less precise domains produce large number of false positives, while analysis with more precise domains doesn't scale on many real-world programs.

Another popular approach is symbolic execution. The main idea of this method is performing analysis by traversing all paths in a function separately. This approach can be used to build a path-sensitive detector, i.e. that can find errors that, at the same time, occur only on a certain feasible function path and are not inevitable for any single point from this path alone. While processing a particular path, the analyzer keeps track of variables values and relationships and computes a path

predicate, i.e. a conjunction of all corresponding branch conditions that are taken along this path. This information is used to prune infeasible paths and check buffer access instructions. Analyzing all paths in a function can be a challenging task due to the path explosion, so a number of techniques are proposed to reduce this problem. A simple, but often effective approach is to abandon the idea of full path coverage and just to stop the analysis after some threshold or time limit reached. Another approach is to merge symbolic states at join points, preserving path-sensitivity of analysis by providing guard conditions for joined states. Third approach, first introduced in Marple, is employing demand-driven analysis [4], [5], i.e. reducing the set of analyzed paths by focusing only on those that end with buffers access.

One of the main obstacles for all mentioned symbolic-execution-based approaches is handling loops. Typical solution is to implement some heuristics to handle the most simple and common loops and ignore other loops. However, there are methods proposed to handle loops with multiple paths inside and summarize their effect on program values [6].

Many buffer overflow errors are caused by violations of function contracts. This can happen when a caller of a library or a user function provides unexpected data to a function, or, on the contrary, a function is not able to correctly handle all input cases implied by the contract. Interprocedural analysis is needed to detect such inconsistencies.

On the lexical analysis level formal and actual arguments matching can be based on similar variables names and usually happens only for the well-known library callees like `memcpy`. For more rigorous scan some tools analyze the whole program as a unified inter-procedural graph. The monomorphic analysis merges information for every call-site — efficient, but imprecise approach. The polymorphic analysis treats each call site individually, so this approach provides context-sensitivity but scales poorly.

An alternative approach is using some approximation of a function's behaviour when analyzing its caller. These approximations can be provided in user's annotations, but they are not always available. A tool can use its own findings obtained by the callee analysis as an approximation. This approach is called summary-based. By choosing the right function order, a tool can minimize the number of missing summaries, but handling recursion still requires additional tricks, e.g. making several analysis passes over strongly connected components of the call graph.

III. BUFFER OVERFLOW DETECTION TOOLS BENCHMARKING

For the past twenty years several studies have been published on evaluating and testing buffer overflow detectors. In addition, there exist different test suites, which provide sets of synthetic buggy and correct code snippets to test the abilities and false positive rate of static analysis tools.

One of the biggest and probably the most popular benchmark is Juliet Test Suite C/C++, created by NSA's Center for Assured Software (CAS) [7]. For C/C++ code it contains

64,099 test cases tagged by CWE entries. Groups corresponding to buffer overflow defects are CWE 121 — "Stack-based Buffer Overflow" (4,968 tests), CWE 122 — "Heap-based Buffer Overflow" (5,922 tests), CWE 124 — "Buffer Underwrite" (2,048 tests), CWE 126 — "Buffer Over-read" (1,452 tests), and CWE 127 — "Buffer Under-read" (2048 tests). Tests in this suite are also tagged with a number called "flow variant" that represents the complexity of control and data flow in a particular test case.

Control flow variants cover different types of conditionals (e.g. `STATIC_CONST_FIVE==5`, `globalReturnsTrueOrFalse()`, etc.) and different control statements (`switch`, `while`, etc.). Data flow variants describe many types of intraprocedural data flow and interprocedural interaction, e.g. data passing through function arguments (via pointer, C++ reference, array, container, etc.), return value, global variable, etc. There are many flow variants that represent C++-specific features and not applicable to C-tests.

We noticed that the distribution of the flow variants is close to uniform in groups of our interest. Another observation is large amount of tests involving wide characters. Many tests contain library function usage, e.g. `memcpy`-like functions, string manipulations, format string processing, etc.

Toyota ITC Benchmark is a test suite created by Toyota InfoTechnology Center aimed at the static analysis tool evaluation [8]. It contains 1,276 simple tests (638 erroneous and 638 correct) divided into 9 types and 51 sub-types. Our interest is in the following tests: sub-types "static buffer overrun" (54 cases), "static buffer underrun" (13 cases) from the "static memory" type and sub-types "dynamic buffer overflow" (32 cases), "dynamic buffer underrun" (39 cases) from the "dynamic memory" type. Each case is represented by a pair of a buggy test and a fixed test.

These samples cover following features in varying combinations: (i) static, stack and heap buffers; (ii) different element types (`char`, `int`, `float`, `struct`, etc.); (iii) index calculations (constant, linear and non-linear expressions, passed as an argument or returned from a function, loop variables and array elements); (iv) obtaining buffer address (local/global variable, function argument, pointer arithmetic including loop variables and aliases); (v) buffer size (heap buffers only with constant sizes, pointer casting); (vi) access types (via index, pointer dereference, in a library function, in a string function).

IV. SURVEY ON OVERFLOW-RELATED CVEs

We believe that although evaluating with a test suite could give a good insight in a particular tool's abilities, any test suite alone can not perfectly represent the whole populations of buffer overflow defects in real code. One (but not the only one) noble goal for static analyzers is to prevent security vulnerabilities to sneak in the project source code. We wanted a better understanding of the features of a static analyzer that are more or less important for achieving this goal. Our survey technique was inspired by [9] and we mostly followed in their footsteps to produce a set of vulnerabilities to classify.

We have to note that detection of exploitable vulnerabilities is not the only goal of a static analyzer. Still there are some types of defects that don't lead to vulnerabilities or may not be exploited with ease, but it is undesirable to have those in the source code. Besides, we believe that nowadays developers more intensively use different (static and/or dynamic) analysis tools before releasing the product. For this reason many simple defects are eliminated during the development process and don't appear in the vulnerability databases. Consequently, we think that analysis of the vulnerabilities can reveal the weakest sides of modern static analysis and show potential improvement directions.

First of all, we have randomly picked 100 entries from the "overflow" category from the CVE database [10]. For 25 of them we could find a source code of the vulnerable version to inspect. For each defect we have studied its causes in the code and then classified the defect by several attributes. Our set of attributes is based on the taxonomy provided in [11] with some changes.

Our first insight is that there are some trends in our sample that can be explained by the source of this sample (vulnerability database): (i) most of the overflows from our sample (72%) happened on write memory access, only few on read access; (ii) only the upper bounds of buffers are exceeded in the defects from our sample; (iii) almost all defects (92%) occurred when tainted data (unbounded data from network, file read, input parameters etc.) overflowed some buffer.

We also noticed that simple errors (e.g. using unsafe functions like `strcpy`) are present in the old code (before 2010), but rarely in the late entries. We believe that this can be partially explained by the usage of code analysis tools.

In our sample about a half of overflowed buffers (48%) reside on a stack, other half (48%) is allocated on a heap, and just a few are global variables.

40% of all defects have overflowed buffer accessed via index (e.g. `buf[i]`), 12% via pointer dereference, 44% via library calls, 24% of which are string functions. The latter requires C-strings modeling to properly analyze such patterns. When buffer is accessed in a library call, we think of size/limit argument as an index (when it's reasonable) for further investigation.

According to our data, 48% of all vulnerable buffers have constant size (all stack and static buffers and a few buffers on the heap). Another 16% have a size that is calculated as a linear combination of other variables. As a result, almost half of all inspected defects require deep analysis of integer variables relationship to detect them.

Another feature that we have evaluated for every entry is whether buffer allocation is global or resides in the same function with buffer access. We have found that this is true only for 24% of defects. On the other hand, all index calculations are in the same function with the access in 32% of defects. Both properties are true for 12% of defects. It follows from the foregoing that interprocedural analysis is essential for buffer overflow detection.

Last thing that we have checked is whether there exists a program point that any path through this point will lead to a corresponding error. If there is no such point, then we assume that path-sensitive analysis is needed to detect this defect. Our sample contains only 28% of defects, for which such a program point exists. This means that path-sensitivity will provide the real advantage for a static analysis tool.

V. SVACE BUFFER OVERRUN DETECTOR

Svace is a static analysis tool that is designed to find as many defects of different types as possible with few false positives and acceptable analysis time [12]. The purpose of this work is to improve the Svace buffer overflow detector with the most needed features. Our detector implements the interprocedural path-sensitive detection algorithm based on symbolic execution with state merging [13]. For now the analysis scope is limited to detection overflows of buffers with compile-time-known size. Our detector looks for faulty paths in a function, i.e. it reports a warning if it finds a path that for any input values is either infeasible or produces an error. Such a strict defect definition is chosen to prevent many false positives caused by unknown function preconditions.

For a buffer access instruction we collect a predicate that implies that there exists a faulty path through this instruction. We use an SMT solver to search a solution for this predicate if any. In case of this formula is satisfiable, we use its model provided by the solver to extract a faulty path. It follows from our experience that simply asking solver for any index value that exceeds buffer bounds in our case leads to many false positives. Reasons for that are unknown function precondition and symbolic path conditions being not precise enough (due to poor loop handling, calls of unknown or complex functions, etc.).

Our interprocedural analysis is implemented using summaries. In the function summary we save the information about relationships between integer values on function entry and exit points. We also save overflow conditions for those input-dependent buffer accesses whose correctness can only be checked in the caller context. Such facts can be propagated to the caller more than once, so the analysis can find an overflow of a buffer allocated in a function that is far away on the call stack from a function with the access instruction. We also implemented a heuristic to handle simple loops that have an inductive variable iterating over an arithmetic progression. Currently on Android 5.0.2 our detector emits 351 warnings with 65% true-positive ratio.

VI. CONCLUSION

We have inspected a number of buffer overflow test suites, related CVE entries, and the source code of large production projects that our tool regularly analyzes. All three sources are essential to understand the design goals of a production quality static analyzer. Test suites expose a set of features to support that is easy to understand, classify, and check. On the other hand, they don't provide a real picture of a production code. Inspecting vulnerabilities is useful, but provides an

exploitation-biased sample. Besides, it does not include defects eliminated during the development process (probably with the help of some static analyzer). Finally, while developing a static analyzer one always deals with false positives produced by the tool and reported by customers, but getting false negative samples is much more difficult. True positives reported by the other tools could be useful, but most of the state-of-the-art tools are proprietary and their results are closed.

From what has been said above it follows that interprocedural analysis, path-sensitivity and loop handling are essential. An analysis can really benefit from tracking affine relations between variables and modeling C-style strings as a very important case of buffers.

Our current goal is to improve the Svac buffer overflow detector to reduce the number of false negatives while preserving the moderate level of false positives. For the aforementioned reasons we think that the most promising directions are handling buffers with dynamic size, C-string modeling, and tracking tainted values. We are working now on the extension of our detection technique described in Section V by tracking string length changes happening during string operations in much the same way as we track buffer indexes while calculating integer values. We believe that this will be sufficient for most of cases, but there are some promising works in the area of string solvers [14] that would additionally allow to track also string contents.

As we have seen, static analysis detection of buffer overflows requires a number of techniques from vastly various fields to move on the road from expectations to real code, and there will always be a way to go.

REFERENCES

- [1] J. Viegas, J. T. Bloch, Y. Kohno, and G. McGraw, "Its4: A static vulnerability scanner for c and c++ code," in *ACSAC*, 2000.
- [2] P. Cousot and R. Cousot, "Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," pp. 238–252, 1977.
- [3] X. Allamigeon, "Static analysis of memory manipulations by abstract interpretation – Algorithmics of tropical polyhedra, and application to abstract interpretation," Theses, Ecole Polytechnique X, Nov. 2009. [Online]. Available: <https://pastel.archives-ouvertes.fr/pastel-00005850>
- [4] W. Le and M. L. Soffa, "Marple," *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, p. 272, 2008. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1453101.1453137>
- [5] L. Li, C. Cifuentes, and N. Keynes, "Practical and effective symbolic analysis for buffer overflow detection," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 317–326. [Online]. Available: <http://doi.acm.org/10.1145/1882291.1882338>
- [6] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen, "S-looper: automatic summarization for multipath string loops," *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pp. 188–198, 2015. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2771783.2771815>
- [7] *Juliet Test Suite v1.2 for C/C++. User Guide*, Center for Assured Software National Security Agency, 9800 Savage Road Fort George G. Meade, MD 20755 - 6738, 12 2012.
- [8] S. Shiraishi, V. Mohan, and H. Marimuthu, "Test suites for benchmarks of static analysis tools," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Nov 2015, pp. 12–15.
- [9] T. Ye, L. Zhang, L. Wang, and X. Li, "An Empirical Study on Detecting and Fixing Buffer Overflow Bugs," *Proceedings - 2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016*, pp. 91–101, 2016.
- [10] "CVE security vulnerability database. Security vulnerabilities, exploits, references and more," <https://www.cvedetails.com/index.php>, accessed: 2018-04-08.
- [11] K. Kratkiewicz and R. Lippmann, "A taxonomy of buffer overflows for evaluating static and dynamic software testing tools," *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, vol. 500, no. November, p. 44, 2006.
- [12] A. Borodin and A. Belevantsev, "A static analysis tool svace as a collection of analyzers with various complexity levels," *Proceedings of ISP RAS*, vol. 27, pp. 111–134, 2015.
- [13] I. A. Dudina and A. A. Belevantsev, "Using static symbolic execution to detect buffer overflows," *Programming and Computer Software*, vol. 43, no. 5, pp. 277–288, Sep 2017. [Online]. Available: <https://doi.org/10.1134/S0361768817050024>
- [14] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 114–124. [Online]. Available: <http://doi.acm.org/10.1145/2491411.2491456>

An approach to simulation-based verification of SoC bus controllers

Mikhail Chupilko*, Ekaterina Drozdova†

*Institute for System Programming of the Russian Academy of Sciences (ISP RAS)

†Lomonosov Moscow State University, Moscow, Russia

Email: chupilko@ispras.ru

Abstract—The paper presents an approach to verification of commutation components of SoC. The core idea is to verify bus controllers and supporting interface parts at unit-level, having reference models written in SystemC. The reference models in the proposed test system is supposed to be easily adjusted to the required bus parameters. The approach has been applied as an in-house prototype to verification of a Verilog model of Wishbone controller. There is a possibility to extend the approach to other busses and protocols by development of a library of supported interfaces.

I. INTRODUCTION

This paper is devoted to the problem of the technology of unit-level verification of commutation parts of HDL-models. Each SoC in fact is an HDL-model, where IP-blocks, being parts of the system, are connected according to some traditional communication protocol (Wishbone[1], OCP-IP[2] or something else). To verify it, one has to somehow obtain a golden model to be referred to in the verification process either to create such a model. In case of IP-blocks, their reference models are typically provided by their vendors. In case of the commutation part connecting IP-blocks, the situation is more difficult. There might be a standard bus controller with a predefined bus width, or, that is more common, there will be an implementation of the standard protocol. The integration problem also looks quite important, as physical layer of bus protocol is not the only thing that can be erroneous, but the incompatible logic of data transfers between different IP-blocks also might be a weak point.

In this research, we propose a technology of unit-level verification of communication models in the case when there is a SystemC reference model provided by a vendor or when this model is absent. C++TESK[3], C++ library of macros meeting all typical requirements of unit-level verification, including reference modeling, stimulus generation, and coverage estimation, is selected as a basic tool for technology definition. This tool supports both reference model development in terms of its macro library and easy attachment to any C++-code.

The rest of this paper is divided into five sections. Section 2 contains more information about communication protocols, including information about Wishbone. Section 3 describes related works in the field of verification. Section 4 presents a proposed approach to unit-level verification. Section 5 studies an example of the approach application. Section 6 discusses the results of the work and outlines directions of future research and development.

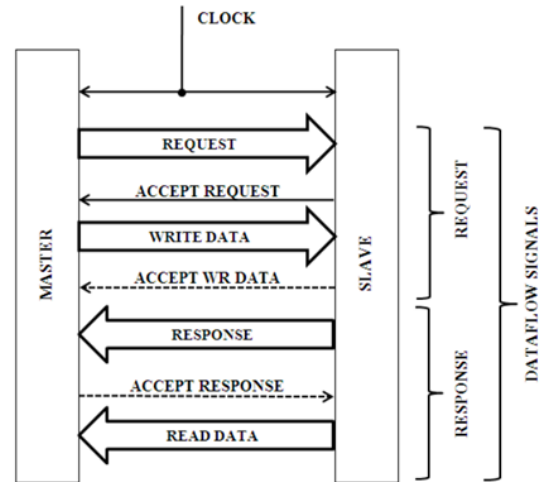


Fig. 1. Open Core Protocol Architecture.

II. COMMUNICATION PROTOCOLS AND WISHBONE STANDARD

As such, each communication protocol is a system of rules allowing several entities to transmit data to each other. More specific definition includes descriptions of possible entities and fixes a physical parameters of transferring. Typically, when speaking about communication part of SoC, one means transmission layer between active (e.g., processor blocks) and passive (e.g., RAM) blocks inside of SoC. To implement the standard interface for data transmission between IP-blocks, a bus controller is used. The aim of the standard interface is to decrease the number of integration problems and support possible re-use.

There are different ways to connect IP-blocks together. The simplest one is a point-to-point connection. As an example of point-to-point connection, let's consider OCP-IP[2] (Open Core Protocol International Partnership, see Figure 1), which is a medium layer between blocks and the bus. OCP is oriented to typical master (active component) – slave (passive component) communication. The protocol was proposed several years ago as a first step in development of a single standard and flexible solution in communication.

Another standard of communication in SoC is Wishbone[1]. Being widely distributed, it was selected in this work for

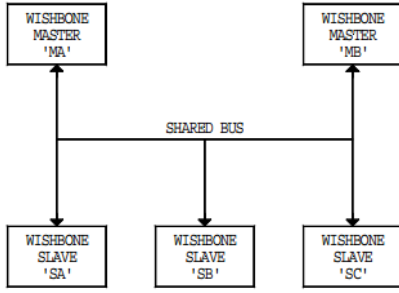


Fig. 2. Wishbone Shared Bus Interconnection.

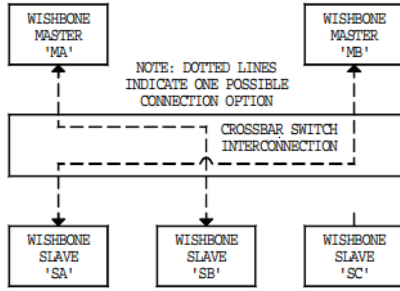


Fig. 3. Wishbone Crossbar Switch Interconnection.

being an example for test system development. The Wishbone standard specifies a standard interconnection between computational IP cores. It supports interconnection of few IP cores using such methods as a point-to-point, a shared bus (see Figure 2, a crossbar switch (see Figure 3, and a switched fabric. The first one represents a simple interconnection between two IP cores where the one called Master initiates the data exchange and another one called Slave responds to this call. The second method supports binding more than two blocks in a consequent order. This method is efficient when the data should be transferred from one IP core to another repeatedly. The third and the fourth methods are similar and also represent the interconnection between several IP cores. In both of them, there is a common bus connecting more than two Master and Slave cores. The Master core can evoke any Slave connected to this bus but the only one at the cycle. Using the crossbar switch method, several Master-Slave cores can be interconnected simultaneously; using the switch fabric — only one pair of cores is connected.

From this review, one can derive the following ideas. First, due to the point-to-point connection being the basics of communication in all cases, when testing the bus controller, IP cores interfaces should be also taken into consideration. Second, the bus controller is limited in its types of requests (mainly, send and receive), the most important for testing situations seem to be with handling of protocol violations and prevent collisions in bus access.

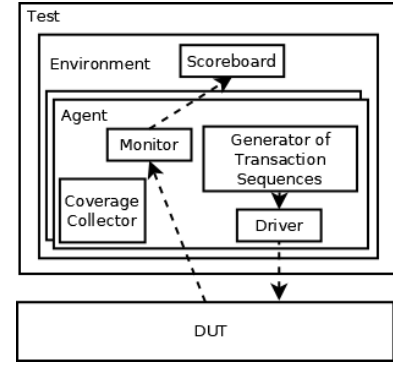


Fig. 4. UVM Test Architecture.

III. RELATED WORK

The task of SoC verification is typically considered as a problem of a mainly unit-level verification. In this case, DUT (Design Under Test) is taken separately from its environment. Stimuli are applied and reactions to check are received via pins of the DUT.

Among unit-level verification approaches, UVM (Universal Verification Methodology) created by Accelera is the most popular. It represents the union of Open Verification Methodology (OVM), Advanced Verification Methodology (AVM) and Universal Reuse Methodology (URM). UVM is a library built upon the SystemVerilog language that provides some basic classes such as the class constructing the testbench structure, the class serving as a basic data structure, the class defining transactions to be passed through components of UVM. This methodology can be used for a constrained random, a coverage-driven, an assertion-based, and emulation-based verification. The UVM testbench structure is as follows (see Figure 4 schematically depicting UVM test). To begin with, there is the DUT. The transaction sequencer block serves to interact with the DUT by generating sequences of bits to be transmitted to the DUT. The monitor block is responsible for listening the communication of the DUT and the sequencer, and gets responses from the DUT. The block called scoreboard compares and evaluates all the information that the monitor is receiving from the DUT and the prediction made by the monitor, describing which output is expected to be taken from the DUT. Sequencers, monitors, and coverage collectors together are called agents. An agent and a scoreboard form the environment. At the same time, there is no any evident method for a making a golden-model as itself, it is up to engineers. The SystemVerilog language is also more a congregation of different methods to describe properties to be checked, rather than a general-purpose programming language convenient for the golden-model development.

Another approach to unit-level verification is C++TESK Testing Toolkit [3], [4], which is represented by a library of C++ macros. The methodology can be used for constrained random and coverage-driven verification. The C++TESK testbench structure (see Figure 5) is similar to the UVM testbench structure but there are some distinctions. There is a stimulus

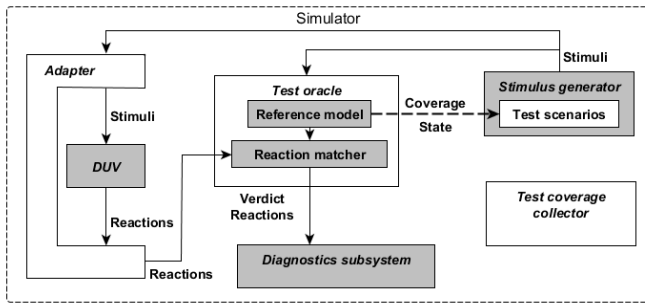


Fig. 5. Verification Environment Architecture.

generator that chooses one of the predefined stimuli to be sent to the DUT and to the reference model expressed explicitly. The comparator gets the output from the model and from the DUT, compares it and evaluates the coverage. C++TESK is compatible with SystemC as a source of reference models, which is one of the key advantages comparatively with UVM.

IV. PROPOSED APPROACH

In this section, we present an approach to functional verification of bus controllers. Each bus controller is either represented by a SystemC model as well as by an RTL description or there is only RTL code and no reference model at all. The conformance between the abstract (SystemC) reference model and the RTL description should be established. This can be done in a simulation-based manner by verifying both sides against the same specification or by verifying the RTL description against the (SystemC) reference model.

To perform verification, a C++TESK Testing Toolkit has been selected, being a C++ library with all necessary classes and macros for specifying design behavior, generating stimuli, and checking reactions. A C++TESK-based verification environment is structured as shown in Figure 5.

The central component of the verification environment is a test oracle, which is responsible for checking whether the DUT behaves properly. It typically includes a reference model that takes stimuli as an input and produces reference reactions as an output, and a reaction matcher that intercepts reference reactions and implementations reactions provided by the DUT and composes reactions pairs. Usually, the reaction matcher works independently for each output interface.

To transform high-level stimuli to the low-level ones and low-level reactions to the high-level ones, the adapter is used. It consists of multiple interface adapters, each being connected with a single input or output interface. An interface adapter describes a simple protocol of putting a single stimulus or getting a single reaction. A special component of the reaction matcher, called a reaction arbiter, specifies ordering between reactions. The task of the reaction arbiter is to choose a reference reaction (if there are any) for a given implementation one. There are two main predefined strategies: model-based arbitration and adaptive arbitration. The first strategy implies that the reference model is accurate enough to predict reaction

order for some output interface (the arbiter selects the next reference reaction stored in the interface buffer). The second strategy is when the reference model is time inaccurate, in which case, given an implementation reaction, the arbiter searches for a reference reaction being equal or similar to the given one. If some reactions are mismatched, a diagnostics subsystem explains what is wrong with the DUT in terms of incorrect, missing, and unexpected reactions.

Other components of a verification environment are a stimulus generator and a test coverage collector. The stimulus generator creates stimuli by exploring the abstract state space of the reference model. The generator is supplied with a set of available stimuli and a function for abstract state calculation; it tries to apply each stimulus in each reachable abstract state. Speaking about verification of a bus controller, it is natural to consider e.g. the number of messages in the bus channel controllers as being the abstract state (though any other abstraction is possible). Such an adjustable stimulus generator allows to produce hard-to-get-into situations which include those with missing messages in one long packet transmission. The test coverage collector estimates the verification completeness basing on user-defined functional coverage metrics, which is more efficient than simply code coverage.

The typical way of C++TESK usage for unit-level verification in case of its own reference model is described in earlier papers (e.g., [3] or [4]), the method of connection to SystemC reference model should be developed. To be more precise, SystemC model should include not only the model of DUT itself, but the model of its environment, including full communication topology. It allows to send complex requests, model collisions, and so on.

The following scheme of commutation between C++TESK and SystemC model is proposed (see Figure 6). C++TESK stimulus generator substitute a master component for the controller bus. Stimuli are applied to both SystemC model (via function calls; to its selected master as if this master wants to send stimuli the generator applied) and to DUT (via procedural interface between C and HDL-simulator; to the input master interface of DUT). SystemC computes the behavior of all environment and creates reactions those are to be got from DUT and checked, and those which are in fact additional stimuli to DUT, e.g. responses from slaves to DUT which are requested by the bus controller and which are substituted by test environment. Output DUT reactions are checked against correspondent SystemC reactions by C++TESK reaction comparator. To provide the program interface between C++TESK and SystemC model, there is a top SystemC class encapsulating all the interfaces between masters and slaves, and model of bus controller. This class is referenced to in C++TESK golden model. In order to register reactions from SystemC in proper C++TESK adapters (serializers for stimuli and deserializers for reactions to be checked), there are special listeners of SystemC model activities (more precisely, bool vector of new reactions on different interfaces). When there is some new reaction, it is registered either to be applied to DUT as a stimulus, or to be added into list of reference reactions to find correspondent

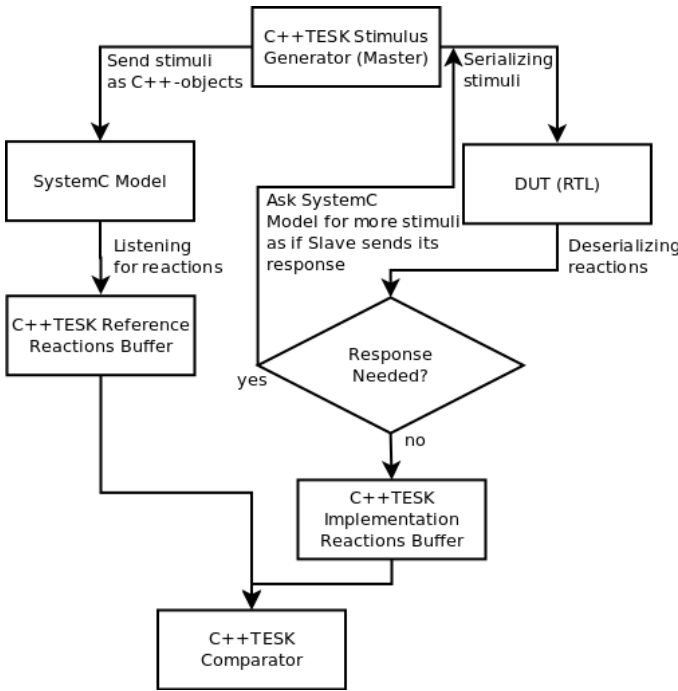


Fig. 6. Architecture of C++TESK, DUT, and SystemC Models Interconnection.

implementation reaction in given time.

V. CASE STUDY

To develop the prototype of verification system and to make experiments with it, a Verilog model of simple Wishbone controller has been taken. The controller supports only point-to-point connection. A correspondent SystemC model has been developed from scratch, taking into account the necessity to support different bus sizes and different topologies of master-slave interconnections. The resulted class (see the following listing to see the main part of it) has been attached to C++TESK reference model as an object that should be stimulated by stimulus generator, and with a possibility of sending reactions that are to be checked against the Verilog model. The whole aspects of the earlier proposed architecture including stimulus generation, HDL implementation reaction checking, and coverage estimation have been kept alive in the prototype of test system. Experiments show that the proposed ideas really work and that future research into this field is required.

```

template <typename adr_bus_size, typename data_bus_size>
SC_MODULE(Controller) {
    typedef Master<adr_bus_size, data_bus_size> master_type;
    typedef Slave<adr_bus_size, data_bus_size> slave_type;

    // types for containers with masters and slaves
    typedef std::map<master_type*> masters_type;
    typedef std::map<slave_type*> slaves_type;

    ...
    SC_HAS_PROCESS(Controller);

    Controller(sc_module_name _name, bus_mode mode,
              masters_type &masters, slaves_type &slaves) :
        sc_module(_name), mode(mode);
  
```

```

// to register all system's masters and slaves
// and to bind all masters and slaves to the controller
void register_master(master_type &master);
void register_slave(slave_type &slave);

// to listen for request messages
void request_listener();
...
}
  
```

VI. CONCLUSION

The approach of communication parts of SoC by means of adjustable SystemC reference models and by C++TESK's stimulus generator, reaction matcher, and coverage collector has been proposed. Description of the approach includes the architecture of test systems. The idea has been checked in form of a test system prototype for a Verilog model of Wishbone controller. The final result of this research is expected to be a creation a SystemC library of adjustable models of widely distributed bus standards.

REFERENCES

- [1] *Specification for the: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Revision: B.3, Released: September 7, 2002.
- [2] *Open Core Protocol Specification 3.0*. Released by Accellera in October 2013.
- [3] M. Chupilko, A. Kamkin. *A TLM-Based Approach to Functional Verification of Hardware Components at Different Abstraction Levels*. Proceedings of the Latin American Test Workshop (LATW), 2011. 1-6 pp. DOI: 10.1109/LATW.2011.5985902.
- [4] M. Chupilko, A. Kamkin. *Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces*. Proceedings of the Model-Based Testing Workshop (MBT), 2013. 67-81 pp. DOI: 10.4204/EPTCS.111.6.

Verification of System on Chip Integrated Communication Controllers

Mikhail Petrochenkov¹, Ruslan Mushtakov², Danil Shpagilev³

Department of Verification and Modeling
MCST

Moscow, Russia

petroch_m@mcst.ru¹, mushtakov_r@mcst.ru², shpagilev_d@mcst.ru³

Abstract —This article presents an approach used to verify communication controllers developed for Systems on Chip developed in MCST. We provide a list of communication controllers developed in MCST and present their characteristics. We describe principles of communication controller's operation, and highlight their similarities. Then we describe a common method of device verification: principles of test system design, test stimuli generation and checking of device behavior. Based on common features of the controllers, we provide the general design of their test system. In addition, we describe specific features of devices that require the adjustments to the common approach. Later we describe how verification of those features affected the design of different test systems. In conclusion, we provide a list of found errors and directions of further research.

Keywords: Elbrus, system on chip, communication controller, Ethernet, DDR4, PCI Express, UVM, stand-alone verification.

I. INTRODUCTION

Modern systems on chip (SOC) may include multiple microprocessor cores, complex hierarchy of caches, peripheral controllers and other types of data processing modules. The task of interconnection between different systems on chip is solved by communication controller (CC) modules. Those modules solve the problem of interprocessor communications, communication between CPU and random access memory (RAM), CPU and peripheral devices, network interfaces, etc. Performance and reliability of communication controllers is crucial for the quality of the whole system. To ensure that communication controllers satisfy all requirements, they must be thoroughly verified. Verification of complex communication controllers is a time-consuming task [1]. One of the widely used approaches to verification of SoC is system verification - execution of test programs (implemented in assembly language) on the model of microprocessor. Another approach is stand-alone verification of SOC components. In this approach, model of the device under verification (DUT) is included in a special program – a test system, which goal is to ensure that DUT satisfies all requirements. This article describes a problem of stand-alone verification of communication controllers with physical media access interfaces in the industrial setting.

The rest of the paper is organized as follows. Section 2 describes communication controllers for physical media access interfaces developed by MCST company. Section 3 presents a common approach to the design a test system and describes its

components. In section 4 we provide a case study for suggested approach applied to specific devices, and adjustments to the approach that were implemented to verify specific features of those devices. In conclusion, we present of verification and provide a direction of further research.

II. OVERVIEW OF COMMUNICATION CONTROLLERS IN “ELBRUS-16C” MICROPROCESSOR

“Elbrus-16C” System on Chip includes many communication controllers. In the following list we will describe ones that require the stand-alone verification: the most complex ones and the ones which reliability is crucial for the functionality of the system.

1. *DDR4 Memory Controller* is a digital circuit that manages the flow of data going to and from the computer's main memory. The controller contains the logical circuits necessary to perform read and write operations in DRAM, with all necessary delays (for example, between reading and writing). The flow of incoming requests is converted into sequences of DRAM commands, while monitoring various conflicts on banks, buses and channels. To increase the effective bandwidth of the memory channel, incoming requests can be buffered and reordered. The reordering mechanism is implemented on the basis of a sequential combination filter system.

2. *PCI Express Root Complex (RC) Controller* transforms packets from in-house protocol to standard PCI Express transaction level packets and implements RC configuration space for communication with peripheral devices. The controller is connected directly to on-chip network to improve throughput and reduce delays. The controller supports up to 16 lanes with speed up to 8 GT/s [2].

3. *Inter-Processor Communication Controller (IPCC)* is designed to solve problems of organization of multiprocessor architectures with shared memory [3]. IPCC functions are logically divided into two levels: the link layer (DLL - Data Link Layer) and the physical layer (PHL - Physical Layer). Exchange by link is carried out by transport packages (containers) of fixed size. Packages contain information about the type of the channel, data, as well as the CRC checksum. Packages are formed into containers according to special rules in order to ensure the priority and maximize the bandwidth of the link. The protocol packets are distributed among several virtual channels (VC) or streams with different priorities. To

ensure the integrity of the data during the transmission over the link, the mechanism of sequential container numbering and CRC encoding are used.

4. *Wide Link Communication Controller (WLCC)* is used to connect south bridge controller to SOC using a protocol similar to PCI Express 2.0 but with reduced overhead. Controller supports memory and configuration space access operations. Supported link width is up to 16 lanes with speed 2.5 or 5 GT/s for each lane. To ensure channel reliability transmitted packets are protected by 16 bit CRC. After transmission, packets are stored in replay buffer waiting for receive confirmation. If negative packet acknowledge is received or time-out is reached, packets are retransmitted. Controller supports up to 8 virtual channels.

5. *10 Gigabit Ethernet Controller* uses 10GBASE-KR interface [4]. It sends and receives Ethernet frames over backplane electrical interface. On a physical layer, it supports procedures of Clause 73 Auto-negotiation and Clause 72 Auto-adaptation. This device supports hardware calculation and checking of Ethernet CRC, IPv4, TCP and UDP checksums, various filtering mechanisms based on MAC addresses and VLAN tags and automatic handling of pause frames.

6. *Gigabit Ethernet Controller* uses 1000BASE-KX interface [4]. Ethernet frames are sent using backplane electrical interface. It supports calculation and checking of Ethernet frame CRC, calculation and checking of IPv4, TCP and UDP checksums, filtering based on mac and IP addresses and automatic handling of pause frames.

Despite the fact that those devices implement sufficiently different protocols, they nonetheless solve a lot of similar problems and implement similar features. Common features of controllers are:

- Register transfer level (RTL) models of this devices are implemented using Verilog and SystemVerilog [5] hardware description languages.
- Controllers communicate with other components on chip using the system interface that implements on-chip communication protocol, and represents transaction layer of the device.
- Controllers don't possess complex internal state and don't implement complex data processing or caching mechanisms. They transform packets between different representations: system level communication protocol packets (used for on-chip communications) and physical interface signals (used for communication on distances beyond the single chip).
- Controllers implement data link layer (DLL) that performs error detection and/or correction using such mechanisms as Cyclic Redundancy Checks (CRC) or forward error correction (FEC).
- Controllers expose the physical interface and implement logical and electrical parts of physical layer for communication with other components on a board. All aforementioned controllers communicate using low voltage differential signaling (LVDS). To ensure clock

recovery and dc balancing devices use physical encoding schemes (for example 8b/10b, 64b/66b, 128b/130b) and signal scrambling.

III. TEST SYSTEM STRUCTURE

Test systems are usually implemented using either general purpose programming languages (C++), hardware description languages (VHDL, Verilog) or dedicated verification languages (SystemVerilog, e, OpenVera). In our company we use SystemVerilog [5] with Universal Verification Methodology [6] (UVM). Use of this language allows for an easy interface with Verilog and SystemVerilog devices, and UVM describes a general test system structure and provides a library of basic verification components.

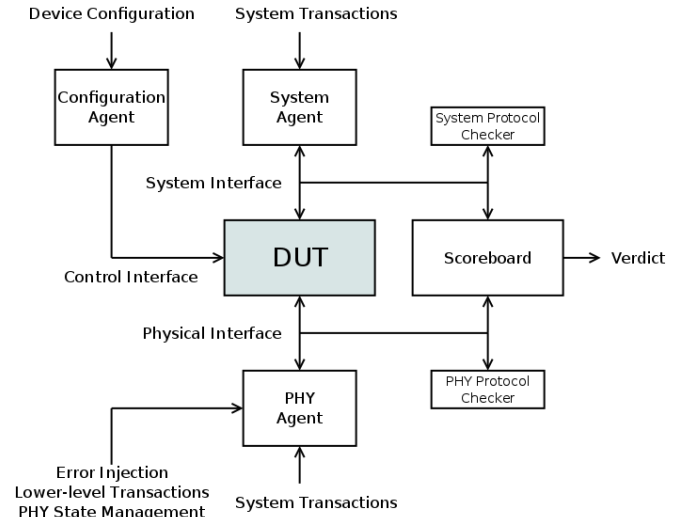


Fig 1. Structure of test system of communication controllers.

Common principles of controller behaviour determine the general structure of the test system. All test systems include a set of basic components.

- Test stimuli generators** are based on constrained randomization. In our case, stimuli generators communicate with system and physical interfaces of DUT. Transactions are described in terms of their attributes and constraints. To specify some test scenario, one must define specific constraints for transactions that will be issued by request generators. SystemVerilog offers a native support for constrained randomization constructs. In addition to transaction transmission and reception, physical agent is able to model some “non-standard” types of behavior: injection of corrupted or non-standard compliant transactions, or handling of received transactions in user-specified way (for example, send negative acknowledge for non-corrupted packet, drop the response to request from DUT, etc..).
- Test system scoreboard** implements a correctness checks. Devices under verification do not possess complex data processing logic and simply perform transformation of transactions between different representations. Scoreboard receives transactions from

system and physical interface monitors and performs comparison between ingress and egress transaction. If discrepancy between expected (transmitted) and received packets is detected, module reports about an error in the test system.

- C. In addition to global test system scoreboard, test system contains **local (system and physical) interface protocol checkers**. Their goal is to check that interface rules and invariants are not violated and otherwise report an error.
- D. **Configuration agent** is used to access a set of memory-mapped configuration registers in the controllers. Those registers are accessed using separate configuration interface. Initial phase of a test is writing desired values to this registers. Test system structure is presented in fig.1.

IV. CASE STUDY

This chapter describes the adjustment and highlights specific implementation details of different test systems.

A. Verification of Link Training and Status State Machine.

One of the features of PCI Express, WLCC and IPCC links is a complex procedure of link initialization and training. During the initialization procedure device sends data patterns containing device capabilities and its current state across the link. Those data pattern are called a training sequence (TS). At the same time, using information from received training sequences, the controller detects the presence of a link partner, determines its active lanes and abilities. Based on this information, pair of devices establishes common mode of operation for transaction transfer. In addition, training sequences are used to change the state of the link (for example, from active to low power mode or to the disabled state).

Presence of the LTSSM provides several additional challenges for the device verification:

- To send the transactions across the link, the active link must be established first. Thus, first action that the controller and its physical link agent partner performs is a link training sequence.
- One must test ability of the device to change its state and check that it reacts correctly to the state change of the link partner.
- In addition to “main” device states there are several “transient” states that the device passes when switching from one main state to another. Depending on training sequences received from link partner in transient states, link training procedure either continues successfully or terminates while reporting the error status.

It should be said that, despite the internal complexity of LTSSM protocols, they are almost invisible to the transaction layer. Only information available to transaction layer is whenever link is currently active or not.

B. Test systems based on a pair of controllers

To verify implementations of in-house communication protocols (IPCC and WLCC) additional type of test system was used [7]. It is based on the pair of RTL-models of communication controllers. In this test systems two controllers are connected using their corresponding physical interfaces. Errors are injected by manipulating the signals of physical interface. The structure of the test system presented in fig.2. Advantages of the approach:

- Simulation of device behaviour in realistic scenarios. Those devices (IPCC and WLCC) use our company’s proprietary protocols are used to connect identical devices, developed in-house. Thus, test system of this kind represents a realistic use-case of the device.
- Simplicity of implementation. The development of physical level agent is a labor-intensive and time-consuming, and its development cannot be avoided by purchasing a third party Verification IP (VIP). In this approach, the development of only a system agent is necessary, and verification can start earlier.

Disadvantages:

- Lower simulation performance is caused by the need to simulate two identical controllers. This doubles the required computational resources.
- More difficult state and error injection control. To inject errors into sent and received transactions one must either directly manipulate external signals of the controller or use hierarchical access to modify the behaviour of the controllers.
- Inability to detect “self-correcting” bugs (for example, incorrect CRC polynomial). This disadvantage is mitigated by the fact those bugs will also self-correct in “real” device.
- Absence of checks on lower protocol levels. The main way to detect an error is to receive an unexpected packet on system interfaces. This may cause difficulties in bug detection and localization in many cases. For example, an error that causes an incorrect request to repeat a transaction can be detected only by performance degradation.

One can reduce the disadvantages while keeping most of some of the advantages of the approach by adding physical monitor on a link between devices.

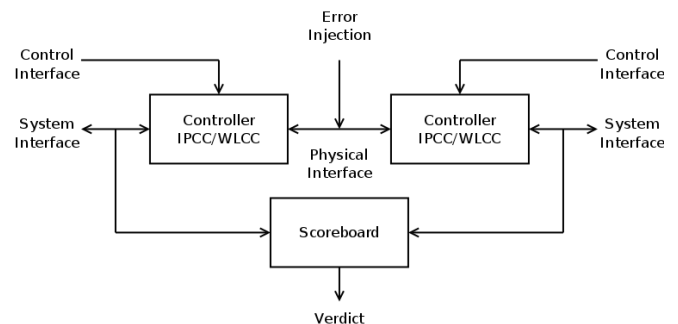


Fig 2. Structure of test system based on a pair of controllers

C. Complex system agent in the Ethernet test systems

Distinctive feature of Ethernet test systems (both 10 Gigabit and Gigabit) is a complex system agent [8]. To reduce CPU usage and increase device efficiency controllers implement Direct Memory Access (DMA). Instead of sending Ethernet frames directly to device interfaces, frames are stored in system memory and the device reads the memory when it is ready for frame transmission. In a same way, the system must prepare a memory space for device to store received frames. The device will write the data to this location after the frame reception. Ethernet controllers are managed using a set of memory-mapped registers. The most important ones are descriptor pointer registers (head and tail). Descriptors contains an Ethernet frame metadata (size of frame, memory location address, higher-level protocol information, etc...). The head register points to the first descriptor available to the controller, and the tail points to the last processed by it. Using those registers the controller reads and writes transaction descriptors and a frame memory. The structure of Ethernet agents is presented in fig. 3.

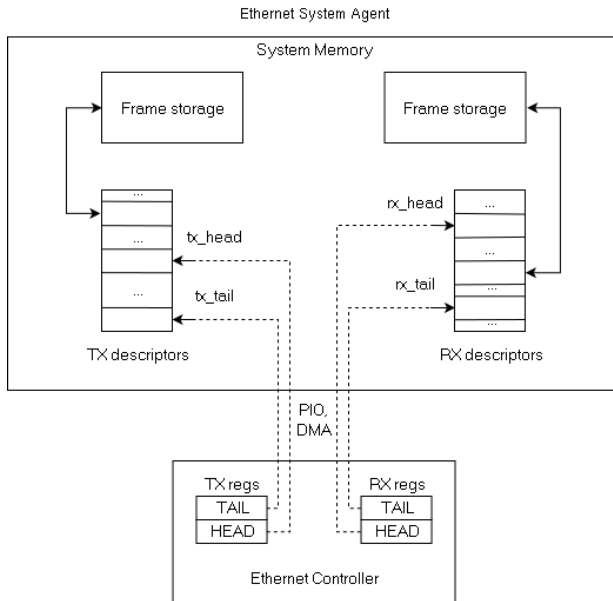


Fig 3. Structure of Ethernet controller test system

D. DDR4 Memory Controller protocol checks.

A system agent in the memory controller test system consists of a set of two modules: the management agent of the information written into the memory and the agent for transferring requests from the system to the controller. The test system requires more sophisticated physical protocol checkers. For this purpose, two modules are used: the DFI protocol verification module and the DDR protocol verification module.

Before active work with the memory is started, the controller performs programming of the operating modes of the DRAM memory modules, conducts its initialization and training. To verify these processes, the DDR Protocol Checker is used. In addition to the fact that the module monitors the initialization and training of the memory, it also controls the execution of all the time constraints imposed to the controller when it issues commands to the memory.

Another important function of the memory controller is to periodically update the data stored in the DRAM using a refresh command. Without periodic updates, DRAM memory chips would gradually lose information, as capacitors storing bits are discharged by leakage currents. DDR protocol checker is used to analyze transactions on physical interface and to check if Refresh commands are issued within specified timing constraints. In addition, the memory state is checked before executing the Refresh command. The memory must be in the IDLE state. The controller has built-in noise immunity mechanisms that allow to check the integrity of the data, and to correct it if necessary. Such mechanisms include: rectification of parity errors of the DDR bus, calculation of checksums, correction of CRC errors on the data bus of the DFI interface while writing, and correction of ECC errors on the DFI data bus during reading. Verification of noise immunity of transmitted data is provided by the DFI Protocol Checker. In addition, checker provides a way to verify the process of switching to and from power saving modes of memory chips by checking their timing parameters.

V. RESULTS

Methods described in the paper were used to verify components of “Elbrus-16C” microprocessor. Errors found in the controllers as a result of stand-alone verification are presented in table 1.

| Device | Number of bugs |
|------------------|----------------|
| DDR4 MC | PCI Express RC |
| PCI Express RC | 48 |
| IPCC | 13 |
| WLCC | 2 |
| 10 GBit Ethernet | 51 |
| Gigabit Ethernet | 22 |

Table 1. Results of stand-alone verification

Verification of those devices is still ongoing. Our future work is aimed at improving those test systems, developing additional test scenarios and using the approach to verify other devices.

REFERENCES

- [1] Stotland I., Shpagilev D., Starikovskaya N. UVM based approaches to functional verification of communication controllers of microprocessor systems. 2016 IEEE East-West Design & Test Symposium (EWDTS)
- [2] PCI Express Base Specification Revision 3.0, <http://pcisig.com/specifications>
- [3] Belyanin I., Petrakov P., Feldman V. Funkcionalnaya organizatsiya I apparatura setevogo vzaimodeystviya modulei v vychislitelnom klastere na baze mikroprocessorov s arkhitekturoi “Elbrus” [Functional organization and hardware means of network interconnection of modules in computer cluster on «Elbrus» microprocessors.] Voprosy radioelektroniki, 2015, no. 3, pp. 7–20
- [4] IEEE Standard for Ethernet. IEEE Std 802.3-2012. 1983 p.
- [5] IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2012

- [6] 1800.2-2017 - IEEE Standard for Universal Verification Methodology Language Reference Manual
- [7] Stotland I., Shpagilev D., Petrochenkov M. Osobennosti funkcional'noj verifikacii kontrollerov vysokoskorostnyh kanalov obmena mikroprocessornyh sistem semejstva "Elbrus" [Features of High Speed Communication Controllers Standalone Verification of "Elbrus" Microprocessor Systems]. Voprosy radioelektroniki, seriya EVT, 2017, 3, pp. 69-75.
- [8] S. Chitti, P. Chandrasekhar, M. Asha Rani. "Gigabit Ethernet Verification using Efficient Verification Methodology". Proc. of International Conference on Industrial Instruments and Control (ICIC), College of Enginnering Pune, India. May 28-30, 2015, pp.1231-1235.

Construction of validation modules based on reference functional models in a standalone verification of communication subsystem

Lebedev Dmitriy¹, Irina Stotland²
Department of Verification and Modeling
MCST
Moscow, Russia
lebedev_d@mcst.ru¹, stotl_i@mcst.ru²

Abstract — Some approaches to functional verification of microprocessor communication subsystems based on developing layered Universal Verification Methodology (UVM) test systems are considered in this paper. The main functions of communication subsystem controllers are transferring and transformation data between microprocessor units. The transformation must be carried out quickly and without data corruption for the correct functioning of the system. Some benefits of application standalone simulation based verification for checking the correctness of communication subsystems are marked out in the paper. Communication controllers could carry additional functions such transmission values of copies of the system registers, address translation and others. The approach of construction a standalone verification environment using UVM is presented in the paper. We also propose some techniques for checking the correctness of communication subsystems which we used to verify the communication subsystem — Host-Bridge — of Sparc V9 microprocessor developed by MCST. The difficulties discovered in the process of test system developing and its resolutions are described. The results of using presented solutions for verification of communicating subsystem controllers and further plan of the test system enhancement are considered.

Keywords — test system, communication controller, Universal Verification Methodology (UVM), reference model.

I. INTRODUCTION

The state of the art in microprocessors composition includes a variety of hardware controllers, which differ in complexity, speed rate, volume and types of data transmitted over them. The characteristics of data are continuously increasing. At the same time, verification costs are increasing, because the possibilities of verification methods are significantly lagging behind the development of microprocessor systems and, accordingly, the correctness checking requires more resources [1].

Each peripheral controller in the system could have its own data format. Converting data format is one of the functions of the interface communication controllers. The communication controllers can be a part of communication subsystem also known as northbridge. The northbridge typically handles communications among the CPU, I/O and in some cases RAM. Therefore, these transformations must be carried out quickly and without data loss. For this reason, the verification of

communication controllers is an important step in the development of the microprocessor system.

The rest of the paper is organized as follows. Section 2 reviews the existing techniques for verifying communication controllers. Section 3 suggests an approach to the problem of developing test system. Section 4 describes a case study and the suggested approaches. Section 5 reveals results and Section 6 concludes the paper.

II. FUNCTIONAL VERIFICATION OF THE COMMUNICATION CONTROLLERS

It is necessary to simulate the operation of entire environment while providing standalone verification of a controller. This requires a test system, the development of which could be started at the earliest stages of whole microprocessor development, as soon as module specification and the RTL-model becomes available. Standalone verification allows detecting errors in the early stages of project. In addition, it helps to create complicated, critical and incorrect situations for the verified module. The achievement of such situations using system verification of the whole microprocessor model takes lot of resources. It is also important to note that the localization of the error is faster, what reduces the debugging time of the controller.

Due to its location between the CPU and the peripheral interface controller, the communication controller, in addition to its basic data format transformation function, could include copies of registers, buffers, FIFOs, parts of distributed control systems, and perform other additional functions. A number of these features should be taken into account in the standalone verification of communication controllers.

It is essential that, according to the classification proposed in [2], the properties of communication controllers include the absence of a pipeline, the absence of strict time (in the system clock frequency) restrictions on transaction processing and tagging of transmitted data. Accordingly, when the devices of this type are verified it is possible to use event-checking modules.

There are a number of methods to build a test system and implement a standalone verification of microprocessor controllers. Among them there is a tool created in the "MCST" named Alone-env, the development of the ISP RAS named C++TESKHW and methodology UVM [3]. The Alone-env tool

simplifies implementation of standalone Verilog tests by creating test sequences in C++. Its library provides a wrapper-class over Verilog description of the verified module. Despite the relative simplicity of using Alone-env tool, there are some disadvantages: the lack of collecting coverage means, high requirements for the testing reference model and the inability to reuse the test system. One of the C++TESKHW tool features is availability of test generation based on the device state graph traversal. However, sometimes it is very hard to define all of the states of device and it needs high accuracy of documentation and checking reference model. UVM is the most widespread verification methodology developed by Accellera Systems [4]. UVM is a library with well-described tools for building portable and reusable testbenches and their components. The test system based on UVM can generate pseudo-random constrained input requests to cover all the possible states of the verified device. Most of well-known simulation tools (like Incisive, VCS, etc.) support the methodology. Moreover, most of VIP (Verification IP) support UVM-based interfaces. We also have a number of test systems and libraries already written and debugged. Therefore, we choose to use UVM for verification of RTL implemented modules of microprocessor systems.

Alone-env and C++TESKHW do not support UVM and we cannot use these tools for UVM-based test system development. UVM provides the universal approach for all types of devices to develop test systems. In this way test systems are becoming more complex and worse in debugging. UVM also has no additional approaches for construction of validation modules based on reference functional models. Therefore, the main purpose of our investigation is to develop and extend the methods of standalone verification of communication controllers using UVM and program reference models.

III. PRINCIPLES FOR THE IMPLEMENTATION OF FUNCTIONAL TESTING COMMUNICATION CONTROLLERS

Standalone verification of communication controllers can be carried out using simulation reference models that are part of the test systems - specially implemented software environment for the verified device. Test system functions includes:

- generating of input requests;
- monitoring of reactions from the verified device and the reference model;
- checking of reactions;
- forming a conclusion about the completeness of testing.

Uvm_sequence_item and uvm_sequence extension classes are defined to generate pseudorandom constrained impacts. The first one defines a set of variables that are required for serialization of set of impacts into a serial bit format. The second performs a single or multiply generating of a set of variables to transmit a request. The request generated by the uvm_sequence_item object is processed by a special uvm_sequencer class and passed to the uvm_driver class. Uvm_driver produces a transformation of generated random requests into sequential bit-vectors in accordance with the interface exchange protocol. Uvm_monitor class is passive. It tracks changes in the interface of the verified device, indicating

the appearance of input or output data, then packages the serial bit signals into the uvm_sequence_item format and transmits for further analysis to the checking blocks. To simplify the structure of the test environment perception, the uvm_driver, uvm_monitor, and uvm_sequencer are combined in the uvm_agent class, shown on fig.1.

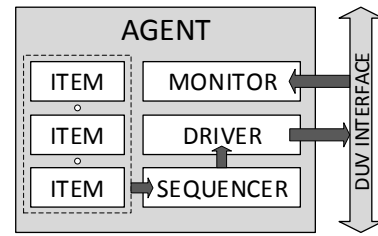


Fig 1. An example of the uvm_agent structure.

Checking the reactions of the verified device can be carried out by internal means of the UVM library, however, if the verified device has a complex structure and many states, the checking module is based on the external to the controller environment reference model usually written in C++. A typical reference model-based test system is shown in Fig. 2.

In Fig. 2 DUV (Design Under Verification) is RTL-model of the verified device, ENV (Environment) - test environment. The number of agents are determined by the number of interface groups of the verified device (tracking the reactions uvm_monitor object can be taken outside from the agents). The reference model generates reference responses when impacts from the test environment are implied. Uvm_scoreboard is a checking module compares the response from the verification device and the reference model and makes a conclusion about the correctness of the operating. Using the DPI (Directed Programming Interface) of System Verilog is necessary to reconcile the types and classes of the test system written in SystemVerilog hardware description language with the C++ language in which the reference model is developed.

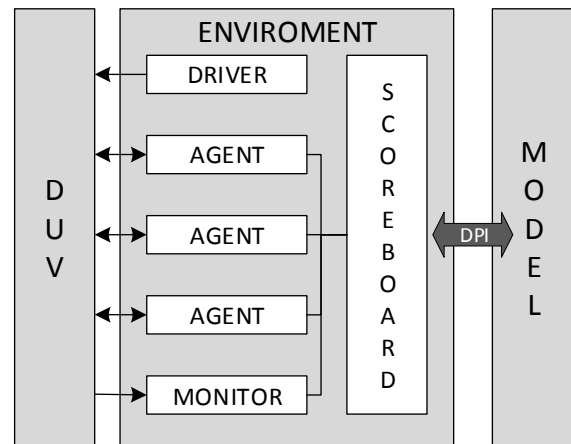


Fig 2. A typical structure of a test system for testing communications controllers.

The reference models could be divided into three types: cycle-accurate, discrete-event with time accounting and event models [5]. The choice of model type depends on the type of verified device, its architecture, and the complexity of

developing a test environment. As stated earlier, the use of event models is justified for communication controllers, as they require less time to develop, maintain changes and can fully simulate the operation of such a controllers.

IV. FUNCTIONAL VERIFICATION OF COMMUNICATION CONTROLLER - HOST-BRIDGE

Host-Bridge is a part of northbridge of microprocessor MCST-R2000 CPU with Sparc V9 architecture developing by MCST. The Host-bridge interface communication controller connects the system with external devices, accepting requests from the system and the I/O space while maintaining the transaction formats accepted in the system and I/O space. The Host-bridge receives requests from the System Commutator, communicate with two I/O channel controllers (IO-links) and provide translation of the virtual address to physical addresses. In addition, controller provides access to the system registers, registers of inter-processor links, memory controller's registers, transferring the new values of the registers to the local copies of them, transmitting interrupts, status signals and collecting snoop-responses. Each type of the registers has its own interface for communication with the destination device. All these features should be taken into account when verifying Host-bridge.

Some approaches for standalone verification using UVM and reference models were observed in [2, 6-8]. In [2] authors used buffers between testbench and reference model for checking marked transactions. In [6, 7] assertions and checking reference model were applied. In [8] there was reference model with very complicated algorithms. In our work, we used model buffers and assertions for checking correctness of transactions. In addition, we present a number of new solutions of the standalone verification process, which was applied for verification of Host-Bridge:

A. Several synchro signals parameters randomization

The main task of communication controllers is to coordinate the requests and data of several devices of the microprocessor system operating at different frequencies of the synchro signal. The parts of the communication controller in which several synchro signals interact should be checked carefully. Generating of random periods of synchro signals and their shifts that are relative to each other can be used for this purpose. The controller specification defines the operating ranges of each synchro signal. At the beginning of each test the frequency and start time of random synchro signal generators are pre-calculated. Thus, it is possible to detect errors in synchronization of internal units: the detection of metastability, desynchronization of releasing requests and data, sticking data in some positions of the buffers.

B. Support of credit exchange mechanisms

To control the flow of requests and free positions in the transaction buffers, Host-bridge supports a credit mechanism, which is a one-bit signal transmission that informs about the availability of free space in the buffers of connected devices. The management of this mechanism allows creating tests with full filling of all positions in the device buffers and needing to wait the vacated space to handle new requests, or on the other hand, tests, when the release of positions provides very fast, and the requests are executed almost instantly. As a result, it is possible

to create test scenarios that are difficult to implement during system testing.

C. Verification of address translation controller

One of the components of the Host-Bridge is the address translation controller IOMMU. It translates a virtual address received from the I/O subsystem requests to physical addresses. The controller sends a request for information about the physical address to a special memory area for providing translation. Virtual address mapping to physical addresses is stored in a special controller buffer – IOTLB (Input-Output Translation Lookaside Buffer). If the buffer is full, the oldest element is displaced. The algorithm of the translation could be represented in the form of several consecutive steps:

- 1) receiving DMA request $p \leftarrow start(x)$,
- 2) analyzing of the input request then matching in the cache IOMMU with the following scenarios:
 - a) match is found (hit IOMMU) – a request with the translated address x'' is executed;
 - b) a match is not found (miss IOMMU) – a request for a physical address x' is executed, then waiting for a response $p.receive(y)$ with the data, and only after that translation of the address is done.

Under dynamic test conditions, there may be situations when a lane in the IOMMU cache is not yet displaced in the RTL-model and the address can be translated without additional request, but it is not present in the reference model. In this case, the reference model will give unnecessary requests to the test system. A global transaction counter was introduced in the reference model to solve this problem. The task of this counter was to identify the source of the requests. In addition, the responses generated by the test system are analyzed too. In the case when the request is successfully translated on the RTL-model side, and the reference model has already given an extra request for a physical address, the test environment generates a response that is marked with a special identifier and sent it to the reference model. When processing a response, the model concludes that the translation was not performed $p.model_check(y)$, calculates the desired transaction identifier $y.id$ and sends it to a special buffer of canceled requests Q_{id} for the physical address. When checking the interchange buffers with reference model after the test completes, the identifier values in the buffer of canceled requests Q_{req} are compared with the identifiers of the remaining unprocessed requests for physical addresses from the reference model. Such remaining requests are not treated as erroneous and are deleted $delete(req.id)$. The pseudo-code of the algorithm is presented below.

DMA handling:

```

while true do
  wait  $p \leftarrow start(x)$ 
  if  $x'$  then
     $p.nccheck(x')$ 
  else if  $x''$  then
    begin
      wait  $p.receive(y)$ 
       $p.model\_check(y)$ 
    end

```

```

     $Q_{id} \leftarrow y.id$ 
     $p.finish()$ 
end
end

```

After test checking:

```

for  $i \in Q_{req}$  do
    if  $c.check(req.id, Q_{id})$  then
         $delete(req.id)$ 
    else  $report(req.id)$ 
end
end

```

D. The correct organization of the exchange in terms of the uncertainty of the issuance of queries

In high-load dynamic tests with many input requests and responses, labeling requests and responses with tags that correspond to positions in the controller's buffers may differ from the values of tags in the reference model. This happens because of the inability to predict time of release of the buffer's position in the event-driven models. Therefore, it is important to match the input and output requests of the model and the verified device. Each request, whether it is an I/O request or a PIO request, has several stages of execution. To ensure the correct functioning of the test scenarios we need to use an associative memory (mappers) for matching. The function of this memory is to store the matching of RTL request tags with reference model tags, when the remaining request data field, such as an address, destination device ID, processor number, and others are compared. Later, when you receive responses to the request, you have to pass to the model the same tag, which was allocated by the model at the stage of forming the request.

Communication controllers in multi-core systems can participate in the coherence protocol and accept snoop responses. Depending on the mode of operation and the content of the fields of the first received for the request response could come as several responses or only one. To complete the request check in coherent mode, it needed to pass all the responses with the correct tags to the model.

E. After test checking

The correct behavior of the communication controller is determined in providing a certain number of responses to requests and receiving the exact number of responses to them. Incorrect operation can be identified by counting the number of received requests, converted to another format requests, and accepted responses. For this purpose, the test system includes transaction counters. They capture all kinds of transactions while the test is running. At the end of the test, special algorithm checks values of these counters and make a conclusion about correctness.

After the test scenario is complete, it also needs to verify absence of unanswered requests in the buffers that link the reference model to the test environment. The presence of such requests signals about error of either the verified device or the reference model.

V. RESULTS

The approaches described above were applied to standalone verification of the Host-Bridge of microprocessor MCST «R2000». Parametrization of synchro signals allowed finding metastability in the controller interfaces. Using different types of credit exchange rate helped to locate deadlocks and livelocks in the controller. Based on one test system the built-in IOMMU controller was also verified. Different configuration of answers with physical address were verified, which helps finding errors in displacement algorithms. After test, checking of model buffers and requests counters provide finding of not released responses to the system.

For the Host-Bridge controller, due to its functional and structural features that belong to the class of communication controllers, a test environment is developed with a checker based on reference event-model. Due to standalone verification of the device 67 errors that have not been found by other means of verification were found and corrected. Code and functional coverage was carried out and 94% coverage was extracted. Gaps in coverage will be eliminated with the further expanding of the test environment with external parts of interrupt system. Total result indicates about effectiveness of standalone verification of communication controller.

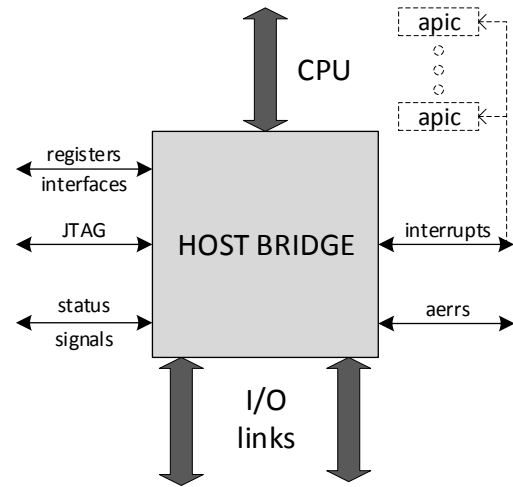


Fig 3. A typical structure of a Host Bridge controller connected with interrupt controllers.

VI. CONCLUSION

Communication controllers are among the important parts of multi-core microprocessor systems have to be thoroughly tested. The principles described in the article do not depend mainly on the implementation of these controllers and allow their full standalone verification. The article suggests the ways to organize the interaction of the test system and the event reference model, as well as ways to resolve the difficulties encountered in the development of the test system.

The proposed approaches have been applied in the verification of the controller Host-bridge as a part of eight-core microprocessor, developed by "MCST". The developed test system and tests made it possible to detect and correct a number of logical errors that were not detected by other test methods.

In the future, it is planned to expand the test environment by adding a part of the interrupt system transmitting signals directly to the core. Figure 3 shows a generalized diagram of a Host-bridge, and the dotted lines illustrate such extension.

REFERENCES

- [1] Kamkin, A.M. Kotsynyak, S.A. Smolov, A.A. Sortov, A.D. Tatarnikov, M.M. Chupilko. Sredstva funktsional'noi verifikatsii mikroprotssessorov [Tools for functional verification of microprocessors]. 2014, pp. 149-199. http://www.ispras.ru/proceedings/docs/2014/26/1/isp_26_2014_1_149.pdf (31.03.2018).
- [2] Shmelev V.A., Stotland I.A. Avtonomnaya verifikatsiya mikroprotssessorov na osnove etalonnykh modelei raznogo urovnya abstraktsii [Standalone verification of microprocessors using reference models with various levels of abstraction]. // Vserossiiskaya nauchno-tekhnicheskaya konferentsiya «Problemy razrabotki perspektivnykh mikro- i nanoelektronnykh sistem (MES)». Sbornik trudov, 2012, no 1, pp. 435-440.
- [3] A.N. Meshkov, M.P. Ryzhov, V.A. Shmelev. Razvitie sredstv verifikatsii mikroprotssessora «Elbrus-2S» [The developement of the verification tools of the Elbrus-2S microprocessor]. // «Voprosy radioelektroniki», ser. EVT, 2014, no. 3, pp. 5-17.
- [4] Standard Universal Verification Methodology <http://accellera.org/downloads/standards/uvm> (31.03.2018).
- [5] Kelton W., Law A. Imitatsionnoe modelirovanie [Simulation modeling] // Klassika CS. 3-e izd. SPb.: Piter, 2004.
- [6] Li-Bo Cheng, Francis Anghinolfi, Ke Wang, Hong-Bo Zhu, Wei-Guo Lu, Zhen-An Liu. A UVM Based Testbench Research for ABCStar. Proc. of IEEE-NPSS Real Time Conference (RT), Padua, Italy. June 6-10, 2016. https://indico.cern.ch/event/390748/contributions/1825090/attachments/1280814/1906413/CR_PosterSession2_268.pdf (31.03.2018).
- [7] Abhineet Bhojak, Tejbai Prasad. A UVM Based Methodology for Processor Verification. Proc. of Design and Verification Conference and Exhibition (DVCON), India. September 10-11, 2015. https://dvcon-india.org/sites/dvcon-india.org/files/archive/2015/proceedings/6_UVM_Based_Processor_Verification_paper.pdf (31.03.2018).
- [8] Kamkin A., Petrochenkov M. A Model-Based Approach to Design Test Oracles for Memory Subsystems of Multicore Microprocessors. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 3, 2015, pp. 149-160. DOI: 10.15514/ISPRAS-2015-27(3)-11.

Medical Images Segmentation Operations

Sabrina Musatian

Mathematics and Mechanics Faculty
Saint Petersburg State University
Saint Petersburg, Russia
sabrina.musatian@yandex.ru

Alexander Lomakin

Mathematics and Mechanics Faculty
Saint Petersburg State University
Saint Petersburg, Russia
alexander.lomakin@protonmail.com

Stanislav Sartasov

Mathematics and Mechanics Faculty
Saint Petersburg State University
Saint Petersburg, Russia
Stanislav.Sartasov@spbu.ru

Lev Popyvanov

Faculty of Medicine
Saint Petersburg State University
Saint Petersburg, Russia
lev.popyvanov@gmail.com

Ivan Monakhov

Faculty of Medicine
Saint Petersburg State University
Saint Petersburg, Russia
ivanishe94@gmail.com

Angelina Chizhova

Faculty of Economics
Saint Petersburg State University
Saint Petersburg, Russia
chilina4@gmail.com

Abstract—Extracting various valuable medical information from head MRI and CT series is one of the most important and challenging tasks in the area of medical image analysis. Due to the lack of automatization for many of these tasks, they require meticulous preprocessing from the medical experts. Nevertheless some of these problems may have a semi-automatic solutions, they are still dependant on the persons competence. The main goal of our research project is to create an instrument that maximizes series processing automatization degree. Our project consists of two parts: a set of algorithms for medical image processing and tools for its results interpretation. In this paper we present an overview of the best existing approaches in this field, as well the description of our own algorithms developed for specific tissue segmentation problems.

Index Terms—deep neural networks, convolutional neural networks, brain tumours, bony orbit, medical images, segmentation

I. INTRODUCTION

Modern ray diagnosis is at the stage of development, and completely different settings and methods are required for different organs: x-ray, MRI, CT, ultrasound are supplemented with invasive contrast methods. Only the doctor can see everything necessary for correct diagnosis and subsequent treatment. However, at the heart of all these methods lie common tasks - the most accurate visualization of the selected zone and obtaining as much data as possible from the results of the examination. In 3D methods (CT and MRI), these tasks are essentially the same, despite the differences in both physical principles and additional settings. Since the ultimate goal of our work is to create a tool that would as accurately as possible visualize isolated structures from raw data obtained by MRI and CT procedures, then this complex work can be decomposed into separate logical components. To isolate complex structures, we formulated the problem of segmentation of tumor processes in MRI images. MRI better visualizes soft tissue and allows to carry out various sequences, change the basic settings of the method in a wide range and use contrast agents. To determine the volume and edge isolation of structures, the problem of determining the volume of bony orbits on a CT was singled out. In this method the bone

structures have a high contrast, the distance between slices is very small, and the method itself is widely distributed and takes little time, which allows to study a large data volume.

From the point of medical informatics those problems are not completely dissimilar and could be solved in a unified manner. Moreover, creating a single instrument that may solve all of these challenging tasks autonomously will not only save doctors time, but also decrease the amount of errors. To the best of our knowledge, there have not been introduced any instrument for automatic segmentation of different body tissues. We came to the conclusion that while the segmentation tasks on different body parts may seem different, they may also all be derived from a core solution based on the deep neural networks.

In this work we explored state-of-the-art solutions based on deep neural networks for brain tumor segmentation and created an ensemble to see if their performance can be improved and used not only for the brain segmentation task but also for complicated head bony structures in general. We use the results of this research as a first step for creating a convenient and powerful instrument for all medical specialities.

II. OVERVIEW

An interest in the possibility of medical images segmentation has increased during the last decade and many different approaches were explored. However, only a few researches evolutionized into a complete useful tools for medicine. Commonly used software, that allows semi-automatic segmentation is Brainlab IPlan (commercial distribution) and ITK-SNAP (open source project) . The main feature of IPlan, that have already been used in several studies [1], [2], is atlas-based segmentation. Atlas is the described and sketched out by experts shape variations of the ROIs(Region of Interest). Due to complexity of human body structure, there are many problems about the accuracy of delineated atlas. ITK-Snap allows segmentation via active contour evolution method - smooth blow-out of preplaced bubbles into the desired region of interest [3]. Although many of the tasks have been solved by these instruments, there are still many problems that specialists

face constantly waiting for improvement. Segmentation is performed by manual or semi-automatic methods.

For the brain tumor segmentation problem many different approaches have been explored and evaluated. There may be formed mainly two classes for these algorithms: methods, which require training on the dataset in advance and those which do not. Early works in this area treated a brain tumor segmentation problem as an anomaly detection problem on the image. Representative works for these approaches may be [4] and [5]. The main advantage of these works is that the presented solutions do not need to be trained beforehand, however that makes it harder to improve the quality of the detection, especially on the smaller tumors. Another class of approaches is based on the idea of using supervised learning methods, such as random forests [6] or support vector machines [7]. These models can learn a powerful set of features and work quite well on the most common cases, but due to the highly discriminative nature of brain tumors it is hard to detect the correct feature set and create a good model. As a result, recent approaches on segmentation refer to the deep neural networks. It is a powerful instrument that has a capability of extracting new features while training and hence may outperform pre-defined features sets of the supervised learning methods. The results of these algorithms may be also used for different kinds of medical images.

We are developing our own tool - Medical Images Segmentation Operations (MISO), which uses neural networks as a back-end for solving various segmentation tasks in medicine. In the next sections we overview separately application of neural networks for brain tumor and bony orbit segmentation as they were trained and used in MISO.

III. BRAIN TUMOR SEGMENTATION

For that task we chose to overview two CNNs(Convolutional Neural Networks) with different architecture which have proven to be the best in this field: DeepMedic [8] - 11-layers deep, multi-scale, 3D CNN with fully connected conditional random field and WNet [9] - fully convolutional neural network with anisotropic and dilated convolution.

A. Data

For the experiments we used BraTS 2017 dataset [10], [11], which includes images from 285 patients of glioblastoma (GBM) and lower grade glioma(LGG). For acquiring this data each patient (Fig. 1) was scanned with native T1, post-contrast T1-weighted (T1Gd), T2-weighted (T2), and T2 Fluid Attenuated Inversion Recovery (Flair). For all patients ground-truth segmentation was provided.

B. Implementation Details

For Wnet we used configuration described in the original papers and BRaTS 2017 dataset for training. For DeepMedic we trained two versions of this network on different datasets and injected some changes into original architecture of this network. For the first version we introduced the following changes: model was trained only on T1 and T2 images.

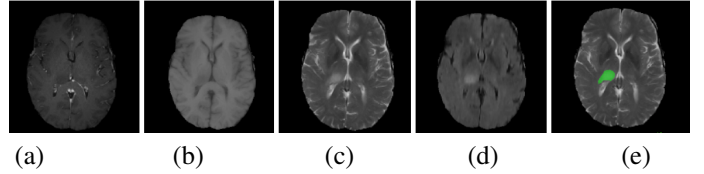


Fig. 1. Original data from BRaTS 2017 dataset: a)T1Gd b)T1 c)Flair d)T2 e)Ground-truth

The reason for that change was that these are the most common MRI sequences. Having a network trained only with this data makes the model available for more hospitals in future. Also, instead of PReLU non-linearity, introduced in the original model, we use SELU [12], which improves the performance and time spend on training. For the second version of DeepMedic we also used SELU, but this network was trained only on T1 images. We wanted to explore how this network will cope when having only one source. For all of these three networks we separated initial dataset into 3 chunks: training(about 80% percent), validation(10%) and test(10%). The performance of these networks on test data may be seen at Table I. In the observed studies authors were aiming not only to detect the tumor but also to segment the tumor into three categories: whole tumor, tumor core and enhancing tumor core. However, in our work we are only interested in the whole tumor detection problem.

TABLE I
INDIVIDUAL PERFORMANCE OF OBSERVED CNNs

| Network | Dice coefficient |
|--------------------------|------------------|
| Wnet | 0.9148 |
| DeepMedic (inputs:T1+T2) | 0.8317 |
| DeepMedic(inputs:T1) | 0.6725 |

C. Detecting the Percentage of False Negative Segments

The original works analyse the quality of CNN performance based on the Dice and Hausdorff measurements, which are good for the segmentation problems in general, but hides the necessary details about misclassifications. For that reason, we explored the results from work of the considered networks to determine the percentage of false positives via false negatives results. Our main goal was to examine whether these methods are more prone to predict false positives then false negatives.

Since the decisive opinion during the diagnosis and treatment is always on doctor, our main goal is to indicate if there may be an pathological tissue and get the surgeons attention to this area. Our system is aiming to find all suspicious areas and send them for reevaluation to medical specialist. Hence, one of the main qualities of this system that should be optimised first-hand would be not false positive results, but false negatives, because those when unnoticed may not get the essential medical care and be a reason for future proliferation of tumor cells. The results of this experiment may be seen at Table II.

TABLE II
NUMBER OF FALSE POSITIVE VIA FALSE NEGATIVE IN THE FINAL SEGMENTATION

| Network | mean (False positive / ground truth) | mean (False negative / ground truth) |
|--------------------------|--------------------------------------|--------------------------------------|
| Wnet | 0.0863 | 0.0830 |
| DeepMedic (inputs:T1+T2) | 0.2330 | 0.1170 |
| DeepMedic(inputs:T1) | 0.4690 | 0.2455 |

TABLE III
THE PERFORMANCE OF NEURAL NETWORK ENSEMBLE. THE RESULTS OF COMBINING NETWORKS TOGETHER DIFFERENTLY

| CNN 1 | CNN 2 | CNN 3 | Dice coefficient |
|--------------------------|---------------------------|-------|------------------|
| Wnet | DeepMedic(inputs:T1 + T2) | - | 0.8861 |
| DeepMedic (inputs:T1+T2) | DeepMedic(inputs:T1) | - | 0.7657 |
| DeepMedic(inputs:T1) | Wnet | - | 0.7941 |
| DeepMedic (inputs:T1+T2) | DeepMedic(inputs:T1) | Wnet | 0.8823 |

D. Neural Network Ensembles

We wanted to detect whether the general performance of these three networks can be improved, when they are used together. So, we proposed the idea of forming the neural networks ensemble [13] out of them. We implemented the following voting scheme: for each voxel we determine each individual result for every neural network, based on their already pre-trained models, and then we qualify a voxel as part of the tumor if and only if the majority of networks classify it as tumor, otherwise it is considered to be a healthy matter. The results of this experiment may be seen at Table III.

IV. BONY ORBIT SEGMENTATION

A. Methods

Our approach consists of two steps. First of all, image classification was presented, dividing initial dataset into two groups: “contains orbit” and “does not contain orbit”. The next step is to segment the orbit in the images marked by the classifier in the previous stage. In this paper first step is described in details, whereas the second step is introduced briefly as it is the subject of further research.

This section describes the approach we have used in details and is organized as follows:

- 1) Data collection
- 2) CNNs model choosing
- 3) CNNs training details
- 4) Output image visualization

B. Data Collection

Raw CT scans was presented by faculty of Medicine of Saint Petersburg State University. Using Toshiba Scanner as instrument and Helical image acquisition as main method, 5 series was made and anonymised. The initial image dimensions were 512×512 , using short(2-byte number) to represent radiation intense with Grayscale Standard display function. Orbits occupy less than $1/4$ of the image, so we reduced the original size from 512×512 to 256×256 in order to decrease computation complexity(Fig 2 b). Slices with orbit was labeled and some of them was manually segmented by expert (Fig. 2 c). Total amount of data: 601 sinus + 80 head CT images were

marked as “contains orbit” and 1414 were marked as “doesn't contain orbit”. 150 images were segmented.

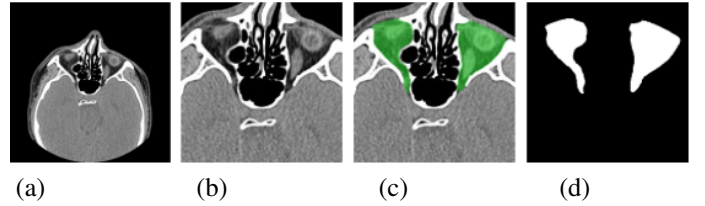


Fig. 2. Data for bony orbit segmentation: a) Initial image b) cropped image c) segmented by expert d) extracted mask (label for cropped image)

C. Model Choosing

To achieve best classification performance of 1st CNN, some important parameters like number of layers and convolutional kernel size must be chosen. So, several kernel sizes and layers number have been evaluated for classification accuracy. The quantitative assessment are shown in Table IV. As a result, the model used for training consisted of eight layers, out of which four were convolutional layers and four were fully connected layers. The output of last fully-connected layer has been fed to a sigmoid function, as it is a standard neural network classification layer [14]. The initial images were cropped and compressed in order to reduce training time. Hence, network accepts grayscale images of dimension 128×128 as inputs. The first layer filters input with 32 kernels of size 5×5 . As it could be seen from experiments, rectified linear unit (ReLU) [15] nonlinearity applied to the outputs of all convolutional layers gives best result compared with other activation functions. The $(n + 1)^{\text{th}}$ convolutional layer takes the output of n^{th} as input processed by ReLU nonlinearity and max pooling layer respectively and process it with $F(n + 1)$ filters. Filters configuration are shown in Table IV. All fully connected layers have equal number of neurons i.e., 256. For the Second CNN the U-net architecture [16] was chosen, as it has already proven its suitability for segmentation in general. Several layer sequences was evaluated to find most fitting model. In order to reduce bias and increase universality, 2 dropout layers with dropout rate equals to 0.2 were added.

TABLE IV
QUANTITATIVE ASSESSMENTS OF DIFFERENT CNN CONFIGURATIONS

| Neurons in each FCLs* | 1st CVL* kernel | Filters model | val.acc. |
|-----------------------|-----------------|---------------|----------|
| 3200 | 11 | 32-64-128-128 | 0.725 |
| 256 | 11 | 32-64-128-128 | 0.9964 |
| 3200 | 7 | 32-64-128-128 | 0.7821 |
| 512 | 7 | 32-64-128-128 | 0.9782 |
| 512 | 7 | 64-64-128-256 | 0.9295 |
| 512 | 11 | 32-64-128-128 | 0.9964 |
| 256 | 7 | 32-64-128-128 | 0.8214 |

FCL - fully-connected layers CVL -convolutional layer, val. acc. - accuracy on validation dataset

D. Training Details

Classification CNN was implemented, trained and evaluated using Python 3.6 as programming language on NVIDIA GTX740M GPU with CUDA Toolkit 9.0 and CuDNN 7.0.5. Keras 2.1.*(version was continuously updated during development) was chosen as neural networks framework, working on top of Tensorflow 1.5*. We have trained and evaluated CNNs on a range different filter models(amount of filters in each convolutional layer), kernel sizes and neuron amount in fully-connected layers. Also experiments with dropout layer [17] were performed.

E. Output Image Visualization

After segmentation has been performed, series of marked images are converted to voxel grid using initial DICOM metadata in order to create 3D model using Marching cubes algorithm by means of MISO Tool and The Visualization Toolkit library. Result is presented in Fig 3.

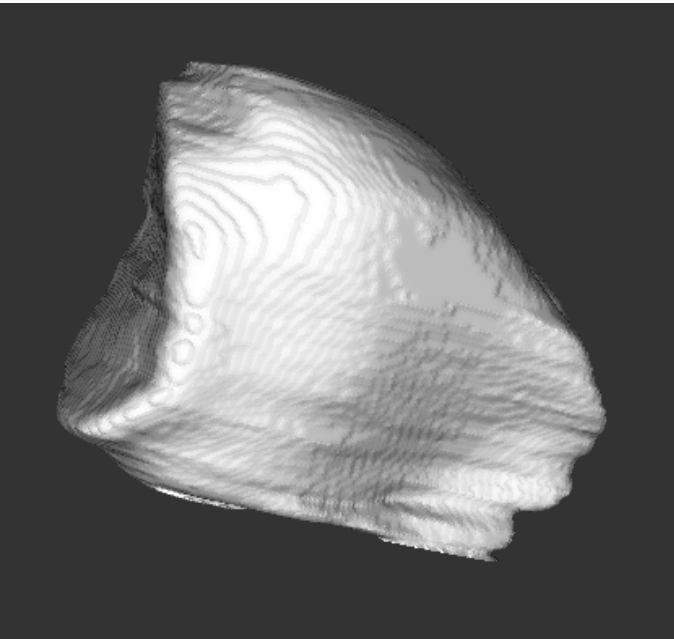


Fig. 3. Rendered bony eye orbit using marching cubes algorithm

F. Experimental Results

1) *Images cropping*: As the main purpose of our work is to create an instrument, that could be run on our servers from multiple clients, in order to deliver the best performance to the customers and lessen waiting time, computation complexity must be decreased as much as possible. To achieve that goal, it was decided to perform experiments with cropped and resized images. When the image was reduced to less than $128 * 128$, we were unable to achieve the required accuracy. The best result under the condition " $accuracy > 0.95$ " showed the approach in which a piece of $256 * 256$ was cut out of the image, which subsequently was compressed to 128. Because of high similarity of head position in CT scans, it was not necessary to move the cropping window.

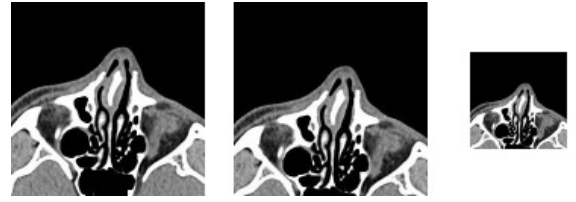


Fig. 4. Different cropping window positions and sizes were examined

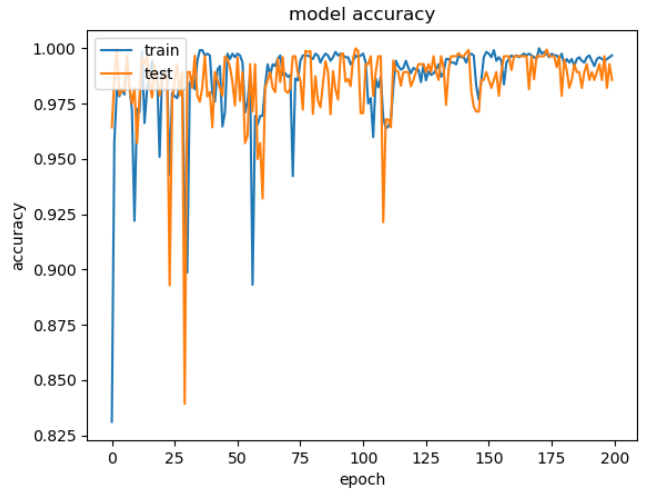


Fig. 5. CNN accuracy on training data(blue) and validation data(orange)

2) *Performance*: For the 1st CNN we used different kernels from 3 to 11 pixels, different CNN model configurations, activation functions and a suitable epoch number to illustrate which one of these properties support CNN to get the highest level of performance. Data was split between train and validation in proportion 4:1. Our model performs best after 115 training epochs - validation accuracy 99% and then stabilizes. Dropout layers with dropout rate lower than 0.4 doesn't impact the accuracy significantly, and more than 0.4 fails the accuracy to 85%, so it was decided to exclude dropout layers from final model. Worth noticing the fact that models with 512 neurons in each FCL showed approximately same result as a model with

256 neurons, but it takes up to 1.4 times more computation time, so 256 was chosen as less resource-consuming.

V. RESULTS

In this paper the first step for the medical segmentation system was introduced. Based on the existing CNN solutions we demonstrated that they may be easily adapted for the segmentation tasks on different medical images. Also, in this work has been shown that these segmentations may be used for creating 3D models and volume estimation. Based on the obtained results, the target tool model was developed using C 7.0 as programming language and .NET 4.7 as framework. As the development is still in the very beginning, there is no purpose for service hosting, although it is considered as the only possible option for the further development, so for now MISO(Medical Images Segmentation Operations) tool has been prototyped as a classic desktop application with CNN results visualization abilities (Fig. 6)

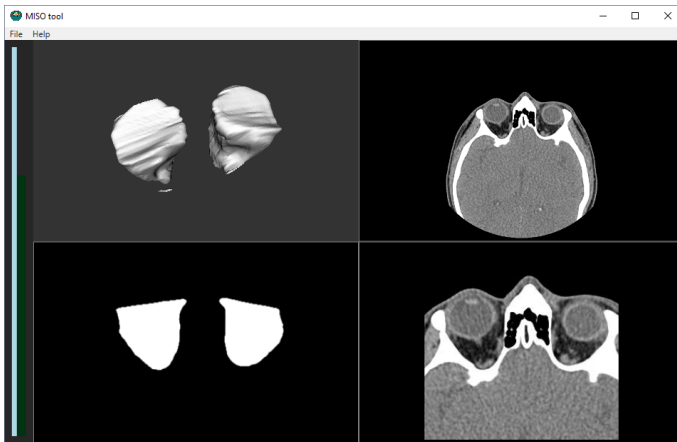


Fig. 6. MISO tool interface

REFERENCES

- [1] Wagner ME, Gellrich NC, Friese KI et al (2016) Model-based segmentation in orbital volume measurement with cone beam computed tomography and evaluation against current concepts. *Int J Comput Assist Radiol Surg* 11:19.
- [2] Jean-Francois D, Andreas B. Atlas-based automatic segmentation of head and neck organs at risk and nodal target volumes: a clinical validation. *Radiat Oncol* (2013) 8(1):111. doi:10.1186/1748-717X-8-154.
- [3] Yushkevich PA, Piven J, Hazlett HC, Smith RG, Ho S, Gee JC, et al. User-guided 3D active contour segmentation of anatomical structures: significantly improved efficiency and reliability. *Neuroimage*. 2006;31(3):1116-1128.
- [4] Doyle, S., Vasseur, F., Dojat, M., Forbes, F., 2013. Fully automatic brain tumour segmentation from multiple MR sequences using hidden markov fields and variational EM. In: *Procs. NCI-MICCAI BRATS*, pp. 1822.
- [5] Cardoso, M.J., Sudre, C.H., Modat, M., Ourselin, S., 2015. Template-based multimodal joint generative model of brain data. In: *Information Processing in Medical Imaging*. Springer, pp. 1729.
- [6] Tumor Segmentation from Multimodal MRI Using Random Forest with Superpixel and Tensor Based Feature Extraction HN Bharath, S Coleman, DM Sima, S Van Huffel International MICCAI Brainlesion Workshop, 463-473.
- [7] Lee, C.h., Schmidt, M., Murtha, A.: Segmenting brain tumors with conditional random fields and support vector machines. In: *CVBIA*. pp. 469-478 (2005).
- [8] Kamnitsas, K., Ledig, C., Newcombe, V.F.J., Simpson, J.P., Kane, A.D., Menon, D.K., Rueckert, D., Glocker, B.: Efficient multi-scale 3D CNN with fully connected CRF for accurate brain lesion segmentation. *Medical Image Analysis* 36, 6178 (2017).
- [9] G. Wang, W. Li, S. Ourselin, T. Vercauteren, Automatic brain tumor segmentation using cascaded anisotropic convolutional neural networks, *Proc. Multi-modal Brain Tumor Segmentation (BRATS) Challenge 2017 - MICCAI workshop*. 2016 (2017). arXiv:1709.00382.
- [10] Menze, B.H., Jakab, A., Bauer, S., Kalpathy-Cramer, J., Farahani, K., Kirby, J., Burren, Y., Porz, N., Slotboom, J., Wiest, R., Lanczi, L., Gerstner, E., Weber, M.A., Arbel, T., Avants, B.B., Ayache, N., Buendia, P., Collins, D.L., Cordier, N., Corso, J.J., Criminisi, A., Das, T., Delingette, H., Demiralp, C., Durst, C.R., Dojat, M., Doyle, S., Festa, J., Forbes, F., Geremia, E., Glocker, B., Golland, P., Guo, X., Hamamci, A., Iftekharuddin, K.M., Jena, R., John, N.M., Konukoglu, E., Lashkari, D., Mariz, J.A., Meier, R., Pereira, S., Precup, D., Price, S.J., Raviv, T.R., Reza, S.M., Ryan, M., Sarikaya, D., Schwartz, L., Shin, H.C., Shotton, J., Silva, C.A., Sousa, N., Subbanna, N.K., Szekely, G., Taylor, T.J., Thomas, O.M., Tustison, N.J., Unal, G., Vasseur, F., Wintermark, M., Ye, D.H., Zhao, L., Zhao, B., Zikic, D., Prastawa, M., Reyes, M., Van Leemput, K.: The multimodal brain tumor image segmentation benchmark (BRATS). *TMI* 34(10), 1993-2024 (2015).
- [11] Bakas S et al. Advancing The Cancer Genome Atlas glioma MRI collections with expert segmentation labels and radiomic features, *Nature Scientific Data*, (2017) [In Press].
- [12] Gnter Klambauer, Thomas Unterthiner, Andreas Mayr, Sepp Hochreiter "Self-Normalizing Neural Networks" arXiv:1706.02515.
- [13] L.K. Hansen and P Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993- 1001, Oct. 1990.
- [14] J. van Doorn, Analysis of deep convolutional neural network architectures, 2014. <http://refereaat.cs.utwente.nl/conference/21/paper/7438/analysis-of-deep-convolutional-neural-network-architectures.pdf>. pages 9, 12, 15.
- [15] Hahnloser RH, Sarpeshkar R, Mahowald MA, Douglas RJ, Seung HS. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*. 2000;405(6789):947-951. doi: 10.1038/35016072.
- [16] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *ArXiv150504597 Cs* (2015).
- [17] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (January 2014), 1929-1958.

Automatic detection of physiologically singular points of the bony orbit

Maria Platonova¹ and Stanislav Sartasov² and Ivan Monakhov³

Abstract. Nowadays it is possible to get a clinically valuable and useful information, for example singular points of the bony orbit from medical images, such as CT. These points are necessary for reconstructive and plastic surgery. Unfortunately, some problems related to these points are still present, such as their vague location description or the influence of human factor when finding them, despite that professional surgeons are dealing with this issue. As a result no fully or semi-automatic tool exists for handling this task.

Therefore, the main goal of this work is to create an automatic tool to accurately determine the location of these points based on convolutional neural networks and deep learning.

1 Introduction

Singular points are important because they are using by ophthalmologists and reconstructive and plastic surgeons. Moreover, the orbital volume, which is an important part of preparation and process of the orbital reconstruction operation, can be calculated with the usage of these points.

According to anthropological data, each point has a strictly deterministic position on the surface of the bone orbit, but in connection with the individual features of the structure of the human skull, some people may have separate position of the same point. Unfortunately, so far there is no such automatic method that would determine the position of these points, while manual operations are prone to human error.

At the present time, medical informatics has a very high level of development, so it became possible to automate some of medical tasks. Substantial information is that this problem can be solved by using computer science methods such as segmentation neural networks applied to CT images. The main purpose of this research is to describe a method to automatically detect the location of singular points in the series of CT images.

2 Singular points

Singular points are special points which located in the surface of the bone orbit. They represent the features of the structure of the bony orbit. According to anthropological data, each point has a strictly deterministic position on the surface of the bone orbit. In connection with the individual features of the structure of the human skull, some people may have separate position of the same point.

These points are located in the whole surface of the orbit: some points are located at the entrance of the orbit, and some located at the inner surface.

An example of a skull model with marked some singular points and their description presented in figure 1:

- point 1 is the vertex of the perpendicular drawn from canalis infraorbitalis;
- point 2 - the point forming with MF the line dividing the orbit in half;
- point 3 - the most medial point of the orbit ;
- point 4 - the middle of the upper edge of the orbit;
- point 5 - the middle of the lower edge of the orbit;
- point 6 - vertex of the orbital opening of the optic nerve canal;
- front-malar-orbital point (fronto-malare-orbitale) - lies at the intersection of the outer edge of the orbit with the frontal-zygomatic suture;
- orbital point (orbitale) - the lowest point of the lower edge of the orbit. It is located on the ocular margin of the malar bone;
- maxillo-frontal point (maxillofrontale) - is located at the intersection of the inner edge of the orbit with the frontomaxillary joint;
- the ectoconchion is the point of intersection of the outer edge of the orbit with a line drawn from the maxillo-frontal point parallel to the upper edge of the orbit;

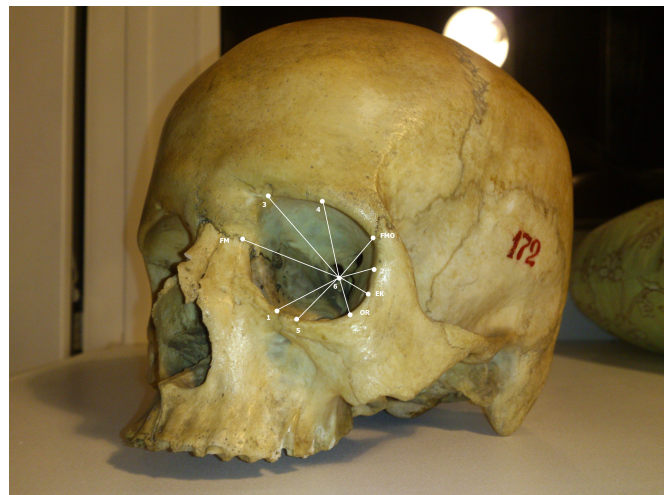


Figure 1. 3D model of human skull with singular points

3 Related works

Applications of the bony orbit singular points are very diverse. As a consequence, an interest to these points increases. Unfortunately,

¹ St. Petersburg State University, email: Platonova.Maria@outlook.com

² St. Petersburg State University, email: Stanislav.Sartasov@spbu.ru

³ St. Petersburg State University, email: ivanische94@gmail.com

there is still a lack of researches and information that could be suitable or helpful during this work.

In addition, some semi-automatic methods are using singular points as one of the main component. As an example, in the article [1] the algorithm was presented for calculating the orbital volume using singular points of the bony orbit, but surgeons manually marked these points.

Therefore, this work is very actual. Moreover, it can be developed for the appliance in other medical tasks that require segmentation as the part of the solution.

4 Description of the experiment

4.1 Dataset

To begin with, there are 5 series of CT images in our disposal. Each series of CT has about 400 images and approximately 140 of them contain an orbit. CT images are provided in a 512x512 Dicom format (Figure 2).

As a dataset for training the neural network, a set of CT images and a set of marked images are used which were prepared by professional surgeon. In addition to the original image a supplementary image containing the location of the singular points is provided. These points are presented as a set of circles with the radius equals 3 pixels that painted with a separate color. Marking these points is a difficult task even for a competent surgeons, so data preparation is a crucial step in this field of research.

To sum up, the dataset was collected and it is consist of 679 CT images that contain an orbit. Moreover, 621 CT image without orbit was added to dataset to provide correct perception of the rest of CT set during the training.

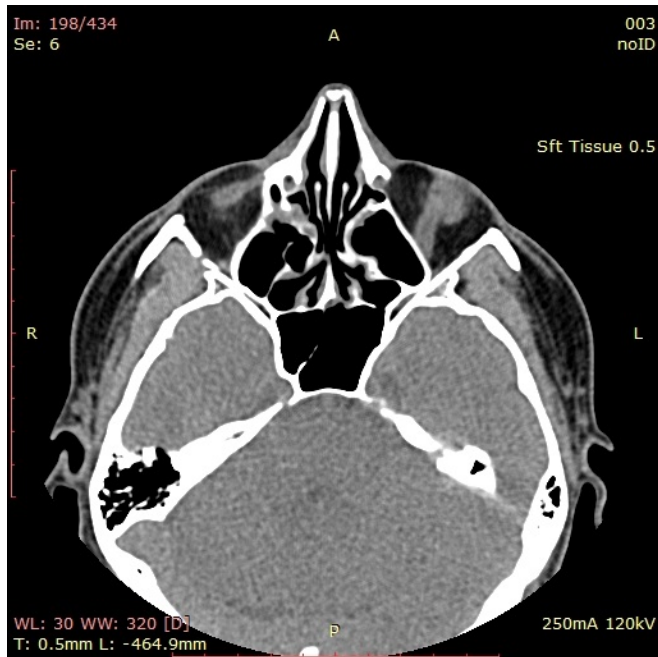


Figure 2. CT image with orbit

4.2 Model

Due to the features of the data, it is difficult to get an extensive dataset, so we decided to use U-net architecture [2], as a mean to

overcome scarce amount of data at our disposal. The U-Net is a convolutional network architecture for fast and precise segmentation of images.

This architecture consists of a down-sampling path, followed by an up-sampling path (Figure 3). In the down-sampling path, max pooling is repeatedly applied to the input image and feature maps. In the up-sampling path, spatial resolution is recovered by performing up-sampling, convolution, eventually mapping the intermediate feature representation back to the original resolution.

During training the input to the neural network will be a CT scan with the corresponding mask, on which singular points are marked as the epsilon ambit. For the experiments, the whole sets of CT images will be used.

In the output layer, sigmoid ($f(x) = \frac{1}{1+e^{-x}}$) is used as the activation function, due to the derivative of the sigmoid function is expressed only through the function itself, that is, no additional computation is needed. The ReLU ($f(x) = \begin{cases} x, & x > 0 \\ a * x, & otherwise \end{cases}$), where a is preset factor, is used in convolutional layers, due to it cuts unnecessary details and is devoid of resource-intensive operations.

The estimation of the accuracy of the neural network process was presented using Dice coefficient as a metrics, since this metric is able to excellent show the accuracy of the segmentation process. The following formula provide the value of this coefficient: $D = 2 * \frac{|A \cap B|}{|A| + |B|}$. The irrationality to use standard metrics for estimating the accuracy was explained as the fact that some incorrectly selected pixels in the area of singular points of the orbit are much more important than the pixels in the rest of the image, even if the recognition was correct.

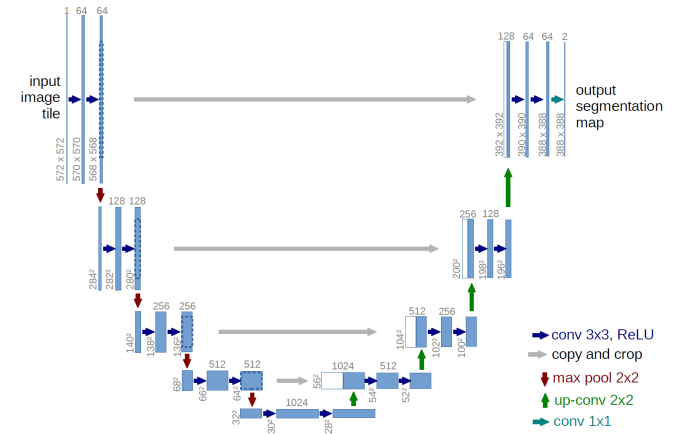


Figure 3. U-Net architecture

4.3 Training

We selected Python as the programming language. Also, we decided to use Keras, working on Tensorflow, as the neural network library. Moreover, as a part of training, dataset was split into 2 parts in proportion 4:1: train and validation. Training of the neural network took place on a service from Google - Collaboration, which provides an opportunity to use the Tesla K80 13Gb GPU for free.

4.4 Result

The results of training are presented on the table 1.

Table 1. Result table

| Number of epoch | Dice coefficient |
|-----------------|------------------|
| 5 | 0,54 |
| 10 | 0,62 |
| 20 | 0.67 |
| 30 | 0.64 |
| 40 | 0.68 |
| 60 | 0.64 |
| 80 | 0.73 |
| 100 | 0.71 |

Moreover, an approbation was carried out using real data, since the result obtained during this testing is more significant for us than the obtained dice's coefficient. The testing took place on 50 CT images. As an example, the figure 4 is shown in which the first part is a CT image, the second part is its mask, the third part is the result of testing.

**Figure 4.** Result of testing

5 Conclusion

In this paper we presented a method for determining singular points of the bony orbit and showed its accuracy using dice coefficient.

It is also worth noting that this method can be used for solving similar problems in fields of medicine and can be expanded for any other part of human body. As a result, plenty of different problems will be able to solve in various fields medicine. For example, in the department of software engineering of St. Petersburg State University similar studies are conducted related to diseases of the stomach, lungs, brain, etc.

To summarize, at the present time, our main trend of applying this method is ophthalmology and reconstructive and plastic surgery. Moreover, we consider that our application to be of great import in medical practice as it may significantly decrease the number of unsuccessful surgeries and diagnostic mistakes.

REFERENCES

- [1] Leander Dubois Thomas J. J. Maal Peter J. J. Gooris Jesper Jansen, Ruud Schreurs and Alfred G. Becking, 'Orbital volume analysis: validation of a semi-automatic software segmentation method', (2016).
- [2] O. Ronneberger, P.Fischer, and T. Brox, 'U-net: Convolutional networks for biomedical image segmentation', (2015).

The Variants of Chinese Postman Problems and Way of Solving through Transformation into Vehicle Routing Problems

Mariia Gordenko

Software Engineering School
National Research University Higher School of Economics
Moscow, Russia
mgordenko@hse.ru

Scientific Advisor: Prof. Sergey Avdoshin

Software Engineering School
National Research University Higher School of Economics
Moscow, Russia
savgdoshin@hse.ru

Abstract— This article provides an overview of the various Chinese Postman problems. The classification of Chinese Postman problems in different types of multigraph (mixed, directed, undirected) is presented. It is shown that the Arc Routing Problems, to which the Chinese Postman problem relate, can be solved with the transformation it into Vehicle Routing Problems. A study of simple algorithms for the solution of a Generalized Travelling Salesman Problem (special case of Vehicle Routing Problem) was carried out. The research is in the process of work. The article also contains plans for future research.

Keywords— *Generalized Routing Problem, Arc Routing Problem, Chinese Postman Problem, Generalized Travelling Salesman Problem*

I. INTRODUCTION

The General Routing Problem (GRP) is a routing problem defined on a graph where a minimum cost tour is to be found and where the route must include visiting certain required vertices and traversing certain required edges [1]. The routing problems are closely related to the logistic and transportation management. From the theoretical point of view, routing problems are mainly related to determining the optimal set of routes in a graph. In practice, the routing problems are not only the tasks of determining optimal set of routes, they are also the tasks of testing robots, the correctness of links in the application menu and operating systems, interactivity usability of web-sites [2]. The Travelling Salesman Problem (TSP) is one of the routing problem consisting in finding a minimal length closed tour that visits each city once. The TSP is one of the most well-known routing problem. Another practical, but less well-known problem is the Chinese Postman Problem (CPP). The CPP is finding a shortest closed path that visits every edge or arc of a graph. The CPP has a simple formulation and a lot of potentially useful applications, but today is poorly understood.

The article gives an overview of various CPPs, provides mathematical formulations of problems, and describes the scope of the problem.

In addition, the article cites references to the literature, in which the various ways of transforming different types of ARP to VRP is described. Also, the results of the current research of various algorithms for solving the problem of a Generalized Traveling Salesman Problem (GTSP) are presented.

II. ROUTING PROBLEMS

A. The General Routing Problem

The general routing is one of the most important problem in the optimization researches [1].

Formally, the GRP is defined on multigraph $G = \langle V, E, A, C \rangle$, where

V is a set of vertices;

A is a multiset of directed edges (arcs);

E is a multiset of undirected edges (edges);

$C: E \cup A \rightarrow R_+$ is a cost function giving non-negative weights of arcs and edges between vertices.

In the routing problems, it is not necessary to visit all vertices, edges and arcs of the multigraph. Two subsets of edges and arcs $A_R \in A$ and $E_R \in E$ are defined. The arcs and edges from A_R and E_R must necessarily appear in the solution. Let the subset of vertices $V_R \in V$ consist of those vertices that must appear in the route.

The goal of all routing problems is to define a minimum cost set of routes, that traverses all the arcs and edges from the multisets A_R and E_R and includes all vertices of the set V_R .

B. The Vehicle Routing Problem

The vehicle routing problem (VRP) is a special case of the GRP with $A_R = \emptyset$ and $E_R = \emptyset$, i.e. the restrictions on the edges and arcs, which must necessarily appear in the route, are absent. The VRP is to determine the Hamiltonian cycle of minimum cost, which traverse all vertices of the subset V_R [3].

In the case, when $V_R = V$, the problem reduces to one of the most famous problem of combinatorial optimization – the classical TSP).

C. The Arc Routing Problem

Another special case of the GRP is the arc routing problem (ARP), it is to determine the minimum cost set of routes, that traverses all required edges E_R and all required arcs A_R of original multigraph [4].

In the ARP, there are no restrictions on the presence of vertices in the route, i.e. $V_R = \emptyset$. The CPP is the variant of ARP.

In the original formulation, the CPP is the problem, where the postman should traverse through every street in the given area.

III. VARIATIONS OF CHINESE POSTMAN PROBLEM

There are a lot of variations of CPP. Below, some of them are described.

A. Windy Rural Chinese Postman Problem

The Windy Rural Chinese Postman Problem (WRCPP) is a special case of ARP, in which $A_R \subseteq A$, $E_R \subseteq E$, and the cost of traversing the edges is depended from the direction of traversing.

WRCPP is a generalization of the CPP in a mixed multigraph. In original CPP problem, it is necessary to find a closed route of minimum length that contains all edges and arcs of the original multigraph at least once. In the real world, it is not always necessary to traverse absolutely all edges and arcs, it is enough to traverse only a certain set of them. Besides, the cost of traversing the edges depend from direction of traversing. The problem of this type is known as the Windy Rural Chinese Postman Problem, which is finding a closed route of minimum length that contains all required edges or arcs of the original multigraph at least once and can contains non-required edges or arcs, so, that the cost of traversing edges depends on traversing direction [5, 6].

Fix the edge $\{v_i, v_j\}$ (non-oriented pair of vertex) from E . Define (v_i, v_j) as ordered pair of vertices, meaning the traversing an edge $\{v_i, v_j\}$ from vertex v_i to v_j vertex. Note, that $(\exists \{v_i, v_j\} \in E) (C(v_i, v_j) \neq C(v_j, v_i))$ (1)

Let arc be $(v_i, v_j) \in A$ ordered pair of vertices, meaning the traversing an arc (v_i, v_j) from vertex v_i to v_j vertex.

We give a formal formulation of the WRCPP problem, extending it to the case of a mixed multigraph.

Let $I = \{1, 2, \dots, |E_R + A_R|\}$, $L = \{1, 2, \dots, |V|\}$. On the set of vertices V of G define indexation $inv = V \rightarrow L$, $(\forall v_i \in V) (\forall v_j \in V) (v_i \neq v_j \Rightarrow i \neq j)$, $i = inv(v_i)$. On the set $E_R \cup A_R$ of G define indexation $inea = E_R \cup A_R \rightarrow I$, $(\forall u_i \in (E_R \cup A_R)) (\forall u_j \in (E_R \cup A_R))$
 $(u_i \neq u_j \Rightarrow i \neq j)$, $i = inu(u_i)$.

The solution of WRCPP is the route $\mu = (v_{l_1}, u_{p_1}, v_{l_2}, u_{p_2}, \dots, v_{l_k}, u_{p_k})$, which satisfy for the following [11]:

- $u_{p_i} = \begin{cases} (v_{l_i}, v_{l_{i+1}}), (v_{l_i}, v_{l_{i+1}}) \in A \\ (v_{l_i}, v_{l_{i+1}}), \{v_{l_i}, v_{l_{i+1}}\} \in E \end{cases} i = 1, 2, \dots, k-1$ (2)
- $u_{p_k} = \begin{cases} (v_{l_k}, v_{l_1}), (v_{l_k}, v_{l_1}) \in A \\ (v_{l_k}, v_{l_1}), \{v_{l_k}, v_{l_1}\} \in E \end{cases}$
- $E_R \cup A_R \setminus \{u_{p_1}, u_{p_2}, \dots, u_{p_k}\} = \emptyset$. (3)

We denote by $C(\mu) = \sum_{i=1}^k C(u_{p_i})$ the cost of traversing the route.

Let \mathcal{M} is a set of WRCPP routes, satisfying (7). It is needed to find $\mu_0 \in \mathcal{M}$, where $(\forall \mu \in \mathcal{M}) (C(\mu_0) \leq C(\mu))$ or $\mu_0 = \arg \min_{\mu \in \mathcal{M}} (C(\mu))$.

A lot of theoretical and computational works is devoted to WRCPP. WRCPP cannot be solved for polynomial time. In general, the problem of WRCPP is NP-hard [12].

B. The Undirected Rural Chinese Postman Problem

The Undirected Rural Chinese Postman Problem (URCPP) is a particular WRCPP which consists of determining a minimum cost circuit on a graph so that it is possible to traverse a given subset of required edges.

DCPP is a special case of WRCPP, where $A = \emptyset$, and there is not edges, which satisfy (1). So, $\forall \{v_i, v_j\} \in E$, $C(v_i, v_j) = C(v_j, v_i)$.

The URCPP is known to be an NP-hard problem and it has some interesting real-life applications.

C. The Undirected Chinese Postman Problem

The Chinese Postman problem in the undirected graph (Undirected Chinese Postman Problem, UCPP) is the original statement of the CPP problem, which was firstly introduced by the mathematician Kwang-Mei-Ko in 1960 [2].

UCPP is a special case of WRCPP, where $A = \emptyset$, $E_R = E$ and there is not edges, which satisfy (1). So, $\forall \{v_i, v_j\} \in E$, $C(v_i, v_j) = C(v_j, v_i)$

If multigraph has Eulerian circuit then this cycle is a solution of UCPP. The algorithm for constructing the Eulerian circuit has $O(|E|)$ time complexity [5].

The Eulerian circuit is existing in an undirected multigraph if multigraph is connected and every vertex has an even degree. A multigraph satisfying the conditions for the existence the Eulerian circuit is called Eulerian multigraph. If the original multigraph is not Eulerian, then for UCPP solution some edges must be traversed more than once. In other words, the multigraph should be supplemented with copies of some the edges to the Eulerian multigraph, so that the cost of the added copies of the edges is minimal [4].

Copies of the edges needed to complement the multigraph to Eulerian can be determined using the following algorithm [6]:

- Find the set of vertices with odd degree $V' = \{v \in V | \delta(v) \bmod 2 = 1\}$. Here the $\delta(v)$ is the degree of vertex v .
- For every pair of vertices $v_i \in V', v_j \in V'$ define the shortest path μ_{ij} .
- In the complete weight graph $G' = \langle V', V' \times V', C \rangle$, where

$$C_{ij} = \begin{cases} C(\mu_{ij}), & i \neq j \\ +\infty, & i = j \end{cases} \quad (4)$$

build matching M with minimum cost. Here the $C(\mu_{ij})$ is the weight of shortest path from vertex v_i to vertex v_j .

- Add copies of edges entering the shortest paths of matching M an undirected multigraph for obtaining an Eulerian multigraph.

The time complexity of the above algorithm is determined by the time complexity of the algorithm for construction a minimum cost matching, which is $O(|V'|^3)$ [7]. So, UCPP belongs to the class of problems P , solved in polynomial time.

D. The Directed Rural Chinese Postman Problem

The Directed Rural Chinese Postman Problem (DRCPP) is a special case of the WRCPP where a subset of the set of arcs of a given directed graph is required to be traversed at minimum cost [2, 8].

DRCPP is a special case of WRCPP, where $E = \emptyset$.

In general, the DRCPP is NP-hard for directed multigraphs [8].

This problem also known as the Selecting Chinese Postman problem.

E. The Directed Chinese Postman Problem

The Chinese Postman problem in the directed graph (Directed Chinese Postman Problem, DCP) is a special case of the WRCPP problem, in which defined on directed graph and all arcs should be traversed. In some articles DCP also called New York Street Sweeper Problem [8].

DCPP is a special case of WRCPP, where $E = \emptyset$, $A_R = A$.

If multigraph has Eulerian trail, then this trail is a solution of DCPP. The algorithm for constructing the Eulerian trail has $O(|A|)$ time complexity [9].

The Eulerian trail is existing in a directed multigraph if multigraph is strongly connected and outdegree of each vertex is equal to indegree. A multigraph satisfying the conditions for the existence the Eulerian trail is called Eulerian multigraph. If the original multigraph is not Eulerian, then for DCPP solution some edges must be traversed more than once. In other words, the multigraph should be supplemented with copies of some the arcs to the Eulerian multigraph, so that the cost of the added copies of the arcs is minimal [2].

Copies of the edges needed to complement the multigraph to Eulerian can be determined using the following algorithm [2]:

- Find the set of vertices with negative divergence $V^- = \{v \in V | d(v) < 0\}$ and the set of vertices with positive divergence $V^+ = \{v \in V | d(v) > 0\}$. Here $d(v) = \delta^+(v) - \delta^-(v)$, where $\delta^-(v)$ and $\delta^+(v)$ indegree and outdegree of vertex v .
- Let the capacity of each arc equal to $+\infty$, determine the minimum cost maximum flow from a set of sources V^+ to a multitude of drains V^- .
- Add an oriented multigraph copies of arcs through which the flow proceeds. In this case, the multiplicity of the added arcs corresponds to the value of the flow passing through the arc.

The time complexity of the above algorithm is determined by the time complexity of the algorithm for construction a minimum cost maximum flow, which is $O(|V|^2|A|)$ [10]. So, DCPP belongs to the class of problems P , solved in polynomial time.

F. Undirected Windy Rural Chinese Postman Problem

The Undirected Windy Rural Chinese Postman Problem (UWRCPP) is an important ARP which generalizes most of the single-vehicle ARP and can be defined as follows [2, 9].

UWRCPP is a special case of WRCPP, where $A = \emptyset$ and there is edges, which satisfy (1).

G. The Undirected Windy Chinese Postman Problem

The Undirected Windy Chinese Postman problem is the NP-hard problem of finding the minimum cost of a tour traversing all edges of an undirected multigraph, where the cost of traversal of an edge depends on the direction [10].

UWCPP is a special case of WRCPP, where $A = \emptyset$ and there is not edges, which satisfy (1). So, $\forall \{v_i, v_j\} \in E$, $C(v_i, v_j) = C(v_j, v_i)$.

If multigraph has Eulerian circuit then this cycle is a solution of WCPP. The algorithm for constructing the Eulerian circuit has $O(|E|)$ time complexity [5]. If the original multigraph is not Eulerian, then some should be traversed more than once. In other words, the multigraph should be supplemented with copies of the edges to the Eulerian multigraph so that the cost of the added copies of the edges is minimal. The solution of the complement problem for a graph that does not satisfy properties (9) and (10) is an NP-hard problem. Thus, WCPP belongs to the class of NP-hard that cannot be solved in polynomial time [13].

H. Mixed Chinese Postman Problem

Mixed Chinese Postman Problem (MCCP) it is a version of WRCPP, where multigraph consists from edges and arcs, simultaneously, and all of them should be traversed [11, 12].

MCCP is a special case of WRCPP, where $A_R = A$, $E_R = E$. and there is not edges, which satisfy (1). So, $\forall \{v_i, v_j\} \in E$, $C(v_i, v_j) = C(v_j, v_i)$.

In 1962, Ford and Fulkerson proposed necessary and sufficient conditions for a mixed graph to be Eulerian. It is necessary and sufficient that in a strongly connected multigraph, the degrees of all vertices are even, and the divergence of each vertex is zero. If a mixed multigraph does not satisfy these conditions, then it must be supplemented by copies of arcs and edges to the Eulerian multigraph, so that the cost of the added copies of the arcs and edges is minimal. The addition of a mixed multigraph to Eulerian is an NP-difficult problem [13].

I. Mixed Windy Chinese Postman Problem

The Mixed Windy Chinese Postman Problem (MWCCP, also called WCPP) is a special case of WRCPP. In MWCCP the cost of traversing the edges is depended from the direction of traversing.

UWRCPP is a special case of WRCPP, where there are edges, which satisfy (1).

In many theoretical works it was shown that problem is NP-hard.

J. Mixed Rural Chinese Postman Problem

The Mixed Rural Chinese Postman Problem (MRCCP) is a special case of WRCPP. In MRCCP not all edges and arcs should be traversed. There is a set of arcs and edges, which must appear in solution, other arcs and edges may appear in solution or may not.

MRCCP is a special case of WRCPP, where there are not edges, which satisfy (1).

In many theoretical works it was shown that problem is NP-hard [14].

We tried to build a classification of different CPP. Combine the existing CPP in a table containing the following criteria:

- the presence of set of edges (A),
- the presence of set of required edges (B),
- the presence of edges with cost, depending on traversing direction (C),
- the presence of set of arcs (D),
- the presence of set of required arcs (E).

The results are shown in Table 1. As we can see, there are four problems, which today are not existing (yellow cells in table), but also can have real-world applications.

TABLE I. THE CLASSIFICATION OF CPP

| | UCPP | URCPP | UWCPP | UWRCPP | DCPP | DRCPP | MCCP | DRUCPP | MWCPP | DURWCPP | URDCPP | MRCPP | DRUWCPP | WRMCCP |
|---|------|-------|-------|--------|------|-------|------|--------|-------|---------|--------|-------|---------|--------|
| A | - | - | - | - | + | + | + | + | + | + | + | + | + | + |
| B | - | - | - | - | - | + | - | - | - | - | + | + | + | + |
| C | - | + | - | + | - | - | - | + | - | + | - | + | - | + |
| D | + | + | + | + | - | - | + | + | + | + | + | + | + | + |
| E | - | - | + | + | - | - | - | - | + | + | - | - | + | + |

IV. SOLVING THE VARIOS CHINESE POSTMAN PROBLEMS

In many sources was shown that almost all ARP problems can be transformed into VRP problems, predominantly in generalized travelling salesman problems (GTSP) [13, 15, 16, 17]. For example, in [16] paper is described how the Capacitated Arc Routing Problem can be formulated as a standard vehicle routing problem. This allows us to transform arc routing into node routing problems and, therefore, establishes the equivalence of these two classes of problems. A well-known transformation by Pearn, Assad and Golden [16] reduces arc routing problem (ARP) into an equivalent vehicle routing problem (VRP). However, that transformation is regarded as unpractical, since an original instance with n required edges is turned into a VRP over a complete graph with $3n+1$ vertices. In [15] article was proposed a similar transformation that reduces this graph to $2n+1$ vertices, with the additional restriction that a previously known set of n pairwise disconnected edges must belong to every solution.

Thus, one can move from less studied problems ARP to well-known problems VRP, such as TSP and GTSP, which have a lot of different approximation algorithms for solving.

In the next sections, we try to compare the simplest algorithms for solving the GTSP.

V. GENERALIZED TRAVELLING SALESMAN PROBLEM

Generalized travelling salesman problem (GTSP) is an expansion of well-known TSP (Travelling Salesman Problem). In GTSP all vertices of graph are grouped in separate clusters. The solution of GTSP is a minimum-cost route, which traverse each cluster exactly once.

GTSP is a special case of GRP, where $E_R = \emptyset$, $A_R = \emptyset$ и $V_R \neq \emptyset$.

Let k clusters are defined V_1, V_2, \dots, V_k , where $V_1 \cup V_2 \cup \dots \cup V_k = V$ and V_1, V_2, \dots, V_k – disjoint sets. Let $I = \{1, 2, \dots, |V|\}$. On the set of vertices define indexation $index = V \rightarrow I, (\forall v_i \in V)(\forall v_j \in V) (v_i \neq v_j \Rightarrow i \neq j), i = index(v_i)$.

It is needed to build a route $s = (v_{i_1}, v_{i_2}, \dots, v_{i_k})$, where each vertex located in different clusters.

Let S is a set of all routes $s = (v_{i_1}, v_{i_2}, \dots, v_{i_k})$ of G .

The cost of cycle $s \in S$ defines as $f(s) = C(v_{i_1}, v_{i_k}) + \sum_{j=1}^{k-1} C(v_{j_i}, v_{j_{i+1}})$, where C – is a cost of traversing from one to another vertex.

It is needed to find $s_0: f(s_0) = \min_{s \in S} f(s)$.

It is NP-hard problem [18].

VI. METHODS FOR SOLVING THE GENERALIZED TRAVELLING SALESMAN PROBLEM

Now, we investigate the following simple approximate algorithms for solving GTSP:

- Nearest Neighbor Heuristic (NN) [19];
- Repetitive Nearest Neighbor Heuristic (RNN) [20];
- Improved Nearest Neighbor Heuristic (INN) [21];
- Repetitive Improved Nearest Neighbor Heuristic (RINN) [22];
- Loneliest Neighbor Heuristic (NLN) [23];
- Double-Ended Nearest and Loneliest Neighbor Heuristic (DENLN) [23].

To evaluate the developed algorithms, the source code was written in the C++ language.

Experiments was conducted on Apple Macbook Pro 13 a1502. Measurements were made of the executing time of the algorithm and the error rate of the solution. The results is presented in Table 1, 2, 3, 4, 5 and 6. $Min(T)$, $max(T)$ and $M(T)$ means minimum, maximum and average time of algorithm working. $Min(C)$, $max(C)$ and $M(C)$ means minimum, maximum and average error rate of algorithms.

TABLE II. THE MEASUREMENTS OF NN ALGORITHM

| $ V $ | $min(T)$ | $max(T)$ | $M(T)$ | $min(C)$ | $max(C)$ | $M(C)$ |
|-------|----------|----------|--------|----------|----------|--------|
| 50 | 0,001 | 0,079 | 0,003 | 6,65% | 23,53% | 14,59% |
| 100 | 0,001 | 0,009 | 0,002 | 7,35% | 21,25% | 15,00% |
| 200 | 0,002 | 0,025 | 0,004 | 8,93% | 21,77% | 15,24% |
| 500 | 0,007 | 0,041 | 0,017 | 12,18% | 35,04% | 20,42% |
| 1000 | 0,025 | 0,114 | 0,062 | 12,99% | 40,77% | 21,33% |
| 1500 | 0,054 | 0,264 | 0,132 | 12,90% | 37,38% | 20,52% |
| 2000 | 0,098 | 3,317 | 0,445 | 12,28% | 39,08% | 20,41% |
| 3000 | 0,350 | 3,888 | 2,510 | 12,81% | 42,00% | 21,29% |

TABLE III. THE MEASUREMENTS OF RNN ALGORITHM

| $ V $ | $min(T)$ | $max(T)$ | $M(T)$ | $min(C)$ | $max(C)$ | $M(C)$ |
|-------|----------|----------|----------|----------|----------|--------|
| 50 | 0,011 | 0,088 | 0,033 | 4,31% | 16,66% | 9,84% |
| 100 | 0,079 | 2,521 | 0,261 | 6,07% | 15,52% | 11,07% |
| 200 | 0,630 | 3,365 | 1,504 | 6,72% | 18,41% | 12,51% |
| 500 | 6,678 | 98,085 | 31,772 | 10,08% | 30,40% | 17,83% |
| 1000 | 62,258 | 695,821 | 247,218 | 11,65% | 37,44% | 18,51% |
| 1500 | 198,014 | 2009,900 | 763,293 | 11,40% | 34,29% | 18,68% |
| 2000 | 489,153 | 6731,020 | 2224,092 | 11,29% | 33,46% | 18,67% |
| 3000 | 4102,820 | 8901,420 | 6334,230 | 12,53% | 38,94% | 20,59% |

TABLE IV. THE MEASUREMENTS OF INN ALGORITHM

| $ V $ | $min(T)$ | $max(T)$ | $M(T)$ | $min(C)$ | $max(C)$ | $M(C)$ |
|-------|----------|----------|--------|----------|----------|--------|
| 50 | 0,000 | 0,008 | 0,001 | 5,81% | 25,32% | 14,50% |
| 100 | 0,000 | 0,009 | 0,001 | 7,48% | 22,43% | 14,79% |
| 200 | 0,002 | 0,009 | 0,004 | 8,43% | 22,36% | 15,27% |
| 500 | 0,011 | 0,054 | 0,025 | 12,57% | 35,96% | 20,30% |
| 1000 | 0,040 | 0,178 | 0,095 | 12,64% | 40,29% | 20,63% |
| 1500 | 0,091 | 0,395 | 0,207 | 12,24% | 38,45% | 20,31% |
| 2000 | 0,155 | 1,035 | 0,404 | 12,09% | 35,56% | 20,24% |
| 3000 | 0,369 | 21,215 | 4,478 | 12,65% | 42,12% | 21,20% |

TABLE V. THE MEASUREMENTS OF RINN ALGORITHM

| $ V $ | $min(T)$ | $max(T)$ | $M(T)$ | $min(C)$ | $max(C)$ | $M(C)$ |
|-------|----------|----------|--------|----------|----------|--------|
| 50 | 0,001 | 0,029 | 0,008 | 6,49% | 22,54% | 14,07% |
| 100 | 0,004 | 0,652 | 0,067 | 7,17% | 20,65% | 14,86% |
| 200 | 0,050 | 1,212 | 0,372 | 8,65% | 23,28% | 15,14% |
| 500 | 0,018 | 0,178 | 0,072 | 12,57% | 35,96% | 20,30% |
| 1000 | 0,094 | 1,177 | 0,446 | 12,64% | 40,29% | 20,63% |
| 1500 | 0,212 | 6,003 | 1,769 | 12,24% | 38,45% | 20,36% |
| 2000 | 0,556 | 28,860 | 6,816 | 12,09% | 35,56% | 20,24% |
| 3000 | 1,007 | 114,590 | 28,321 | 12,65% | 42,12% | 21,20% |

TABLE VI. THE MEASUREMENTS OF NLN ALGORITHM

| $ V $ | $min(T)$ | $max(T)$ | $M(T)$ | $min(C)$ | $max(C)$ | $M(C)$ |
|-------|----------|----------|--------|----------|----------|--------|
| 50 | 0,001 | 0,029 | 0,003 | 6,38% | 22,21% | 14,44% |
| 100 | 0,004 | 0,046 | 0,011 | 8,03% | 20,63% | 14,76% |
| 200 | 0,017 | 0,078 | 0,036 | 8,45% | 21,76% | 15,41% |
| 500 | 0,078 | 0,434 | 0,190 | 12,52% | 34,46% | 20,36% |
| 1000 | 0,293 | 2,262 | 0,814 | 12,65% | 40,48% | 20,68% |
| 1500 | 0,656 | 15,192 | 3,043 | 12,81% | 36,84% | 20,17% |
| 2000 | 0,356 | 19,953 | 3,764 | 12,59% | 38,07% | 21,32% |
| 3000 | 1,456 | 120,110 | 27,402 | 13,34% | 39,91% | 22,33% |

VII. SUMMARY

This article provides an overview of the known CPP. An attempt to systematize and classify these problems has been made. Mathematical formulations of new types of CPP was founded. The paper also shows that almost all problems of the

ARP can be transformed to VRP. In addition, for solving the Chinese Postman problems the way of transformation it into VRP (mainly in GTSP) has been chosen.

At this stage, the research is not complete. It is necessary to investigate the various ways of transformation ARP is into VRP. In addition, it is necessary to investigate the various ways of solving the GTSP. And the key idea of future research is the use of transformation algorithms and algorithms for solving the GTSP for solving the different modifications of CPP.

TABLE VII. THE MEASUREMENTS OF DENLN ALGORITHM

| $ V $ | $\min(T)$ | $\max(T)$ | $M(T)$ | $\min(C)$ | $\max(C)$ | $M(C)$ |
|-------|-----------|-----------|--------|-----------|-----------|--------|
| 50 | 0,001 | 0,015 | 0,003 | 5,22% | 28,44% | 15,82% |
| 100 | 0,005 | 0,088 | 0,012 | 7,65% | 22,59% | 15,73% |
| 200 | 0,018 | 0,089 | 0,039 | 9,71% | 23,78% | 15,97% |
| 500 | 0,019 | 0,192 | 0,058 | 13,14% | 38,21% | 22,05% |
| 1000 | 0,101 | 1,081 | 0,433 | 12,94% | 48,55% | 22,28% |
| 1500 | 0,299 | 22,092 | 2,174 | 13,31% | 39,38% | 21,64% |
| 2000 | 0,440 | 28,828 | 4,095 | 12,59% | 38,06% | 21,31% |
| 3000 | 1,498 | 147,260 | 35,357 | 13,34% | 39,90% | 22,31% |

REFERENCES

- [1] Eglese R., Letchford A., General Routing Problem. In: Floudas C., Pardalos P. (eds) Encyclopedia of Optimization. Springer, Boston, MA. 2008.
- [2] Thimbleby, H. The directed chinese postman problem. Software: Practice and Experience, 2003, 33(11), pp. 1081-1096.
- [3] Toth P., Vigo D. (ed.). The vehicle routing problem. – Society for Industrial and Applied Mathematics, 2002.
- [4] Hertz A., Laporte G., Mittaz M. A tabu search heuristic for the capacitated arc routing problem //Operations research. – 2000. – T. 48. – №. 1. – C. 129-135.
- [5] Zerbino D. R., Birney E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs //Genome research. – 2008. – T. 18. – №. 5. – C. 821-829.
- [6] Edmonds J., Johnson E. L. Matching, Euler tours and the Chinese postman //Mathematical programming. – 1973. – T. 5. – №. 1. – C. 88-124.
- [7] Kolmogorov V. Blossom V: a new implementation of a minimum cost perfect matching algorithm //Mathematical Programming Computation. – 2009. – T. 1. – №. 1. – C. 43-67.
- [8] Robinson H. Graph theory techniques in model-based testing //International Conference on Testing Computer Software. – 1999. – T. 1. – C. 999.
- [9] Wilson R. J. An eulerian trail through Königsberg //Journal of graph theory. – 1986. – T. 10. – №. 3. – C. 265-275.
- [10] Ababei C., Kavasseri R. Efficient network reconfiguration using minimum cost maximum flow-based branch exchanges and random walks-based loss estimations //IEEE Transactions on Power Systems. – 2011. – T. 26. – №. 1. – C. 30-37.
- [11] Chen W. H. Test sequence generation from the protocol data portion based on the Selecting Chinese Postman algorithm //Information Processing Letters. – 1998. – T. 65. – №. 5. – C. 261-268.
- [12] Aho A. V. et al. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours //IEEE transactions on communications. – 1991. – T. 39. – №. 11. – C. 1604-1615.
- [13] Dror M. (ed.). Arc routing: theory, solutions and applications. – Springer Science & Business Media, 2012.
- [14] Ghiani G., Improta G. An algorithm for the hierarchical Chinese postman problem //Operations Research Letters. – 2000. – T. 26. – №. 1. – C. 27-32.
- [15] Longo H., De Aragão M. P., Uchoa E. Solving capacitated arc routing problems using a transformation to the CVRP //Computers & Operations Research. – 2006. – T. 33. – №. 6. – C. 1823-1837.
- [16] Pearn W. L., Assad A., Golden B. L. Transforming arc routing into node routing problems //Computers & operations research. – 1987. – T. 14. – №. 4. – C. 285-288.
- [17] Laporte G. Modeling and solving several classes of arc routing problems as traveling salesman problems //Computers & operations research. – 1997. – T. 24. – №. 11. – C. 1057-1061.
- [18] Fischetti M., Salazar González J. J., Toth P. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem //Operations Research. – 1997. – T. 45. – №. 3. – C. 378-394.
- [19] Solomon M. M. Algorithms for the vehicle routing and scheduling problems with time window constraints //Operations research. – 1987. – T. 35. – №. 2. – C. 254-265.
- [20] Modares A., Somhom S., Enkawa T. A self-organizing neural network approach for multiple traveling salesman and vehicle routing problems //International Transactions in Operational Research. – 1999. – T. 6. – №. 6. – C. 591-606.
- [21] Cheung K. L., Fu A. W. C. Enhanced nearest neighbour search on the R-tree //ACM SIGMOD Record. – 1998. – T. 27. – №. 3. – C. 16-21.
- [22] Tao Y., Papadias D., Shen Q. Continuous nearest neighbor search //VLDB'02: Proceedings of the 28th International Conference on Very Large Databases. – 2002. – C. 287-298.
- [23] Pimentel F. G. S. L. Double-ended nearest and loneliest neighbour: a nearest neighbour heuristic variation for the travelling salesman problem //Revista de Ciências da Computação. – 2011.

Applying the methods of system analysis to teaching assistants' evaluation

Ekaterina Beresneva

Faculty of Computer Science
National Research University Higher School of Economics
Moscow, Russia
eberesneva@hse.ru

Mariia Gordenko

Faculty of Computer Science
National Research University Higher School of Economics
Moscow, Russia
mkgordenko@edu.hse.ru

Abstract— This article presents the results of applying various methods of system analysis to evaluation of teaching assistants. The article shows the process of interaction of teaching assistants with students and faculty. Selection and analysis of criteria for the evaluation of training assistants are carried out. In the article various soft methods for decision-making are considered. In addition, the application of the methods AHP and Fuzzy AHP type-2 to evaluate teaching assistants is considered. In the process of work, the best teaching assistant is selected, and the group of the best teaching assistants is defined.

Keywords—system analysis, combination of soft and hard methods, multicriteria decision making (MCDM), AHP, type-2 fuzzy sets, Fuzzy AHP

I. INTRODUCTION

At the Higher School of Economics (HSE) there is a program “Teaching assistant” which has been effective for several years. Each teacher can invite an education assistant, who will take some of the routine tasks related to teaching the course (checking home work, developing test materials, etc.).

Every student or a graduate student of the HSE, who meets the criteria established by the faculty, can be a teaching assistant. The teacher (or group of teachers) formulates tasks for the teaching assistants and monitors the quality of their performance. The teacher is responsible for the results of the students' knowledge, the quality of materials prepared by the education assistant, methodical support of the teaching assistant's work.

At the moment, all faculties establish their own criteria for selecting teaching assistants independently. Now there is only one criterion for all disciplines: “A student must have a mark at least 8 on the course in which he/she is involved, or he/she must have a recommendation from the department, to which teaching of this discipline is fixed.” However, the practice shows that it is not enough to have only this criterion. There were no special studies about it before, but annual evidence showed that an excellent mark does not fully correlate with being a good teaching assistant. Recent year revealed that 60% of assistants were not able to cope with their work according to teachers. Most problems were connected with personal qualities, professional and communicative skills. For example, somebody did all the tasks slowly and did not do everything in time, or just did not have enough knowledge in the subject area. There were even some facts of disclosure of confidential information: one teaching assistant shared answers to the tests with students. Thus, there is a strong necessity to define a group of selective factors in a clever

manner.

Recently, the head of Computer Science faculty has ordered each teacher (or group of teachers) on all disciplines to choose the best teaching assistant to give him/her an incentive award. In addition, next year the number of students is reduced, and it is necessary to decrease the number of assistants. Now there is a tendency on “Discrete mathematics” course that the education assistants who come from year to year are the same. This situation prompted the idea that at the moment when assessing teaching assistants, it is worth using additional criteria that will allow the group of teachers to select the best assistant and choose the group of the most successful assistants.

Thus, two tasks are faced – to choose the best assistant on “Discrete mathematics” course and to select the group of the most successful assistants, with whom it is possible to continue working on this course.

The purpose of this work is the development of searching method, which will select the best assistant and select the group of the most successful ones according to the criteria set by the group of teachers.

The rest of the paper is organised as follows. We discuss the problem specification in Section 2 and introduce our premises for model, which we use to illustrate our main results on Section 9. Sections 3, 4 and 6 present the different methods used for solution the problem. In sections 5 and 7 the derivations for the AHP and Fuzzy AHP are discussed. Section 8 presents a sensitivity analyse.

II. THE DIFFERENCE BETWEEN PREVIOUS WORKS AND OUR APPROACH

The literature review show there are a lot of researches that reveal a high success of applying the teaching assistant program in general. The most recent one is [3]. However, no one article is aimed neither at selection criteria for teaching assistants nor at searching methodology.

The closest study to our problem is devoted to a proposed framework for evaluating student's performance [4]. This work is based on the hard approach only. It uses the variation of the most widely used approach for multi-criteria decision making – Analytic Hierarchy Process that combines mathematics and expert judgment. Since Analytic Hierarchy Process suffers from the problem of imprecision and subjectivity, their paper proposes to use Fuzzy AHP instead of traditional method.

However, there is an opinion about useless of applying Fuzzy

AHP method. In [3] it is said that “the numerical representation of judgments in the AHP is already fuzzy” and “making fuzzy judgments more fuzzy does not lead to a better more valid outcome and it often leads to a worse one.”

Our article proves that Fuzzy AHP with type-2 modification can still be used in a decision making process. Moreover, our study combines both hard and soft approaches because this problem consists of not only main criteria but also it has a lot of additional ones. And these auxiliary factors can not be described using only formal algorithms.

III. PROBLEM DEFINITION

The problem of finding the best teaching assistant and the group of teaching assistants is closely related with searching the criteria by which the teaching assistants should be selected. To analyze the domain and determine its boundaries, the rich picture can be applied. Rich Picture is a collection of sketches, pictures, photos, symbols, signatures which represent a particular situation or a question (system/problem/need) of the real world from the point of view of the person or group of people who create it. Image components are people (stakeholders), systems, processes, interfaces, data streams, information sources, infrastructure objects, attendant and impeding factors, emotions, points of view and attitude to them, etc.

Rich Picture can reflect the interaction and connections of

the system components (or the surrounding world), their influence, cause and effect. It can also represent such subjective elements as attitude (perception), point of view, prejudice [1].

It is used to explore and aggregate the physical, conceptual and emotional aspects of the actual situation (system/problem/need).

Rich picture on subject “Teaching assistants” interactions in discipline “Discrete mathematic” is provided in Figure 1.

To analyze the subject area and project boundaries, the CATWOE technique is a good addition to Rich Pictures.

CATWOE is defined by Peter Checkland as a part of his Soft Systems Methodology (SSM). It is a simple checklist for thinking. CATWOE is an acronym, each letter stands for a specific word: Clients, Actors, Transformation, World view, Owner, Environmental constraints [2].

The Table 1 shows the result of applying the CATWOE analysis to the domain problem.

After analyzing the processes and interactions associated with the members of the system, a clear understanding of the subject area is emerged. There are three teachers: one lecturer (the leading teacher) and two seminarists at “Discrete mathematics” course. They compose a decision group for choosing best assistants. Fair and reliable evaluation results would be obtained by this group because its members have a strong relationship with teaching assistants during the course.

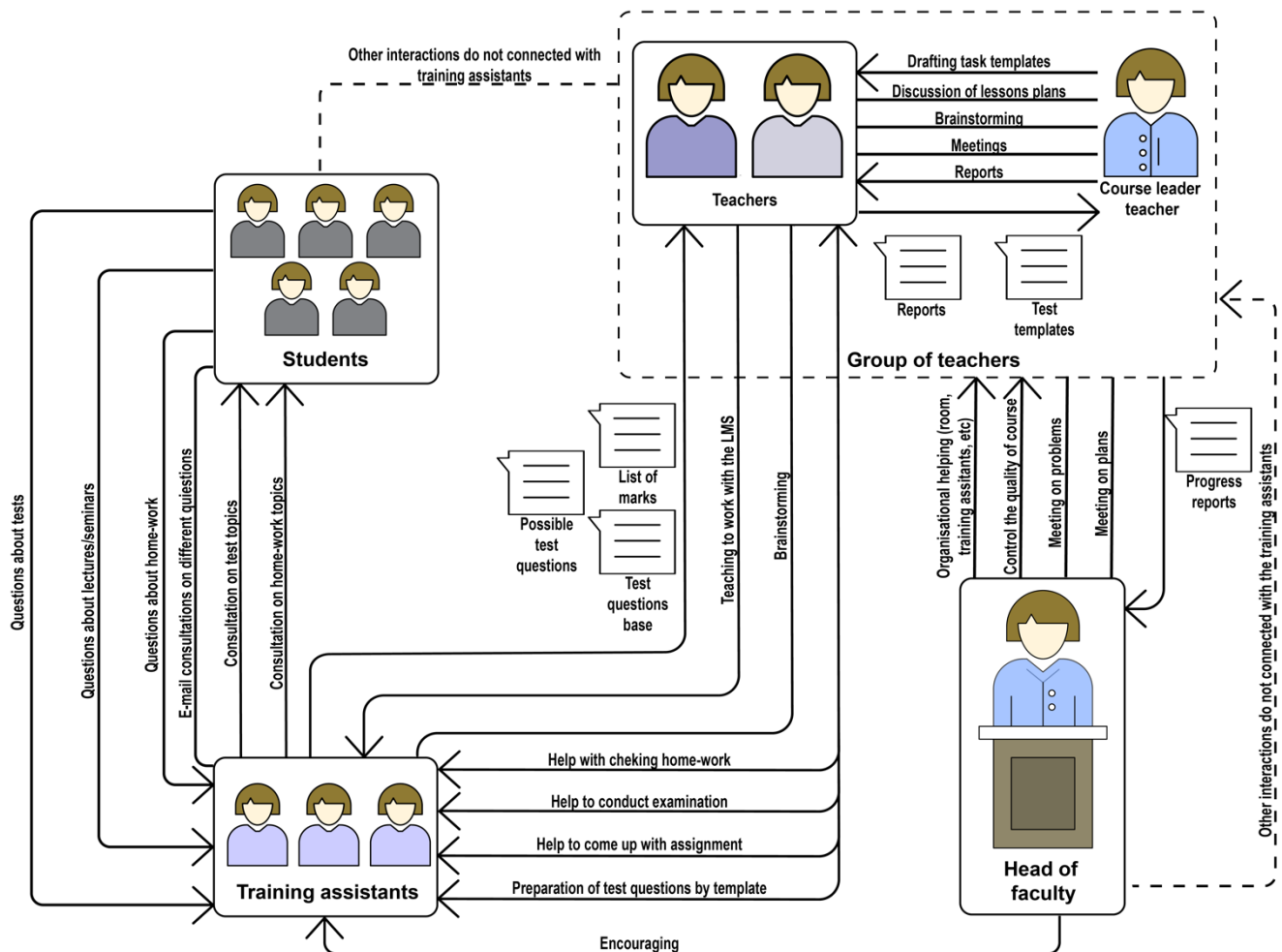


Figure 1. Rich picture on subject “Teaching assistants’ interactions in discipline “Discrete mathematics”

Table 1. The CATWOE analysis

| Role | Description |
|----------------------------------|--|
| Clients | Teachers who want to assess their teaching assistants. Students who need assistants' help. |
| Actors | Groups of teachers who interested in evaluating of skills of teaching assistants and choosing the group of the best teaching assistants. The head of faculty who wants to encourage the best teaching assistant. |
| Transformation | Each teaching assistant receives points for certain evaluation criteria. |
| World View | It is needed to define a group of the best teaching assistants and the best teaching assistant. The definition of a group of best teaching assistants is necessary in order to reduce the risks associated with incompetent and disinterested teaching assistants with the next year group of teaching assistants. |
| Owner | Teachers and the head of faculty. |
| Environmental constraints | National educational and assesment standarts |

In order to evaluate the assistants, it is decided to come up with evaluation criteria. After the first brainstorm, the list of criteria is similar to a chaotic list of records. The next meeting of the teachers shows that some of the criteria identified in the first stage for assessing the assistants turned out to be duplicated or unnecessary. After long discussions and joint brainstorming, three main groups of criteria are identified: professional skills, communicating skills, personal qualities.

The professional skills include the following sub-criteria:

- active involvement in the process of forming the program of discipline;
- initiative to compile new types of tasks for tests;
- knowledge of the subject domain;
- quality of homework checking;
- speed of homework cheking;
- experience of active use of the LMS;

The communicating skills include the following sub-criteria:

- pedagogical experience, the ability to correctly present information;
- openness to student issues (e.g. quick response to questions, competent answers);
- participation in counseling sessions before the tests and examinations;
- active communicating with teachers, participation in weekly meetings;
- the ability to listen carefully.

The personal qualities include the following sub-criteria:

- ethical compliance;
- punctuality;
- self-motivation, the desire for development;
- responsibility for work;
- teamwork skills;
- subordination;
- striving to achieve common results;

- resistance to conflict situations;
- the ability to generate new and innovative ideas;
- the ability to compromise;
- benevolence.

From the first group the next criteria are deleted:

- active involvement in the process of forming the program of discipline. *The teachers should do it, because drawing up a discipline program requires experience and entails a great responsibility.*
- knowledge of the subject domain. *Taking into account that each assistant is selected among the best students of the course, this requirement should be fulfilled by default.*

And the next criteria are combined as they characterize the cheking of homework and are closely interrelated:

- quality of homework checking;
- speed of homework cheking.

From the second group the next criteria are deleted:

- pedagogical experience, the ability to correctly present information. *This ability can be learned. One of the goals of the Teaching Assistant'program is the development of teaching skills.*
- the ability to listen. *In our opinion, this parameter is almost impossible to estimate.*

From the third group the next criteria are combined, because they are very interrelated and cannot be separated:

- self-motivation, the desire for development;
- responsibility for work;

And the next criteria are deleted:

- teamwork skills. *It is related with the responsibility of work criteria.*
- ability to be subordinate. *By default, the main person on the course is the teacher. This is necessary to understand at first.*
- striving to achieve common results. *It is related with the responsibility of work criteria.*
- resistance to conflict situations. *It is the responsibility of the teacher to resolve and prevent the emergence of conflict situations.*
- the ability to generate new and innovative ideas. *This is not a paramount task of the teaching assistant. And the teaching assistant can work great, but do not come up with ideas, it's not scary.*
- the ability to compromise. *The last word for the teacher.*
- benevolence. *It is related with the ethical compliance and punctuality of work criteria.*

Table 2. The involvement in educational process

| | A | B | C | D | E | F | G | H | I | J |
|------|---|---|---|---|---|---|---|---|---|---|
| 1.1. | 5 | 5 | 5 | 5 | 4 | 5 | 3 | 2 | 1 | 1 |
| 1.2. | 4 | 5 | 5 | 4 | 4 | 4 | 4 | 4 | 1 | 4 |
| 1.3. | 1 | 2 | 1 | 4 | 5 | 1 | 1 | 1 | 1 | 1 |
| 2.1. | 4 | 3 | 3 | 4 | 2 | 5 | 5 | 1 | 3 | 1 |
| 2.2. | 5 | 4 | 4 | 4 | 3 | 5 | 2 | 1 | 2 | 2 |
| 2.3. | 4 | 4 | 3 | 4 | 1 | 4 | 5 | 2 | 1 | 2 |

The final elected criteria and subcriteria are shown in Figure 3. All the criteria and subcriteria have their own identification numbers

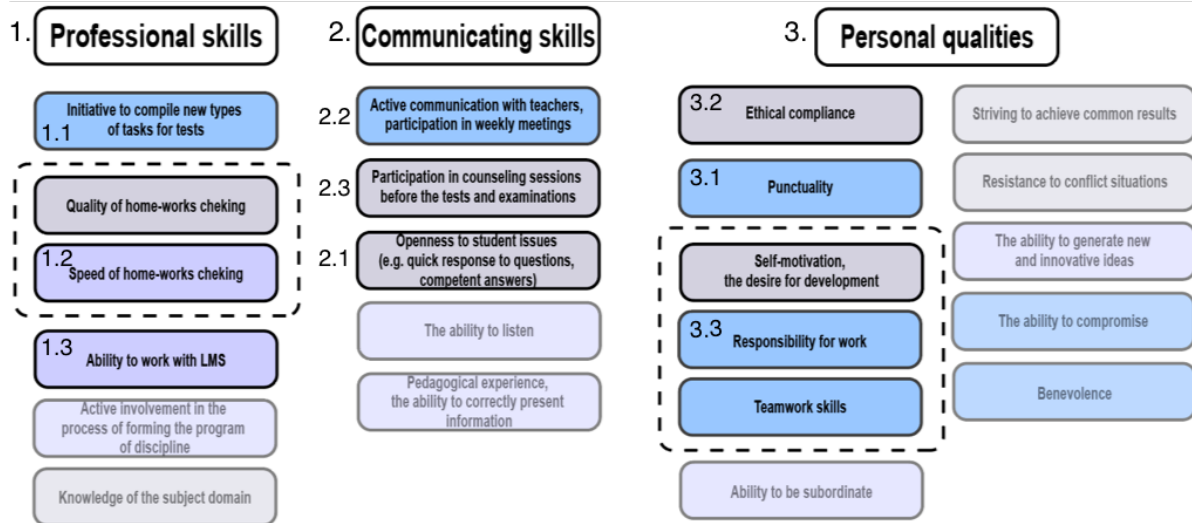


Figure 2. The final list of criteria

IV. EXPLORING THE ALTERNATIVES

There are ten teaching assistants *A, B, C, D, E, F, G, H, I, J* on the course.

We can reduce the number of evaluating teaching assistants after assessing the involvement of teaching assistants in educational process.

We have 3 groups of criteria, consisting of 9 sub-criteria. In order to assess the involvement of assistants in the educational process, we did not use the values of the last three subcriteria (3.1-3.3). These sub-criteria refer to a group of personal qualities and can not be regarded as involvement in the educational process. Then the involvement of the teaching assistant in the educational process for each criterion is evaluated, based on expert judgment. The results are presented in Table 2.

Let's understand which assistants are least involved in the work process, according to experts. Calculations of threshold equals to 3, 4 and 5 are shown in Tables 3, 4 and 5.

Table 3. Threshold is equal to 3

| | A | B | C | D | E | F | G | H | I | J |
|------|---|---|---|---|---|---|---|---|---|---|
| 1.1. | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1.2. | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1.3. | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2.1. | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 2.2. | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2.3. | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

Table 3 and 4 allow to identify teaching assistants who are least involved in the educational process.

Table 4. Threshold is equal to 4

| | A | B | C | D | E | F | G | H | I | J |
|------|---|---|---|---|---|---|---|---|---|---|
| 1.1. | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1.2. | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1.3. | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2.1. | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2.2. | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2.3. | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

The Table 5 with threshold equals to 5 shows that no one from *H, I, J* is not indispensable.

Table 5. Threshold is equal to 5

| | A | B | C | D | E | F | G | H | I | J |
|------|---|---|---|---|---|---|---|---|---|---|
| 1.1. | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1.2. | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1.3. | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2.1. | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2.2. | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2.3. | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |

Thus, it is decided not to consider further the last three teaching assistants (*H, I, J*). However, little involvement in the educational process has its own explanations:

- *H* was ill two month;
- *I* was out of connection;
- *J* decided to switch to another faculty. Preparation for the exams took all his spare time.

Thus, seven candidates are remained. It is difficult to find the best one because each of them is successful in one or more criteria.

Stakeholders are about to choose *A* as a winner because this assistant took part in all teacher meetings and he suggested new types of tasks for tests so regularly (approximately once every two weeks). Assistant *A* communicated with teachers a lot (flushed before their eyes), that's why they prefer him.

However, this decision can be too unfair, that's why multicriteria decision making (MCDM) process is decided to be applied.

V. ANALYTIC HIERARCHY PROCESS

Analytic Hierarchy Process (AHP) which is one of the most used MCDM approaches [6] is a structured multicriteria technique for organizing and analyzing complex decisions including many criteria. In this paper we use a classical AHP proposed by the author [7].

At the first step of AHP a model for the decision is developed. Experts break down the decision into a hierarchy of goals, criteria, sub-criteria and alternatives (see Fig. 3).

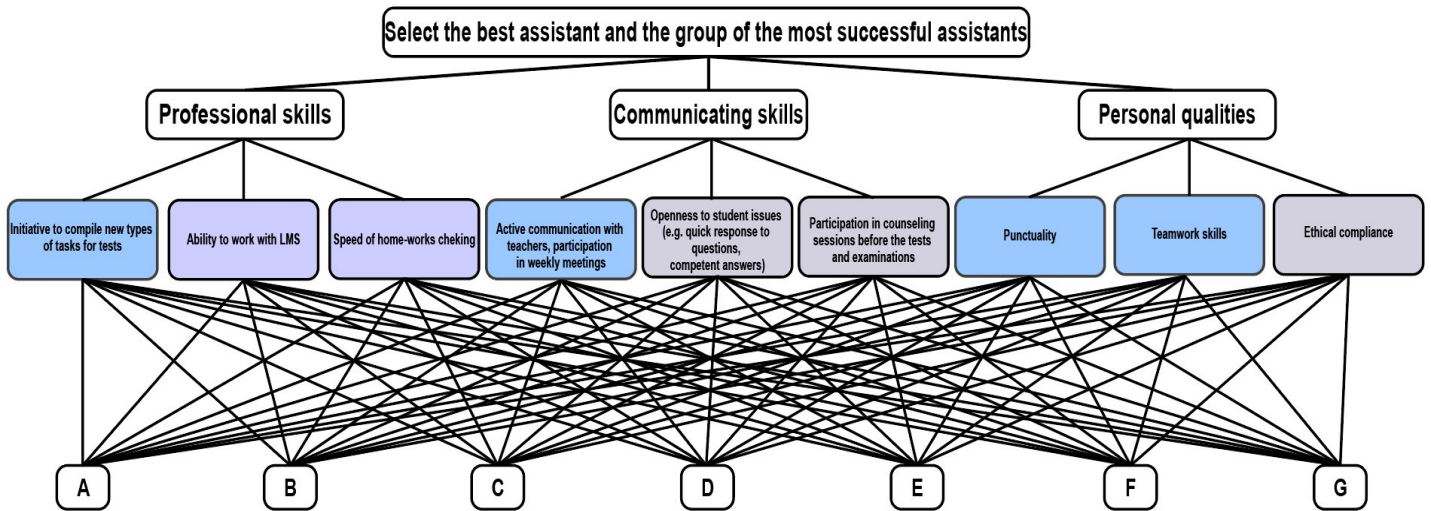


Figure 3. Decision model

After that, decisioners derive priorities (weights) for the criteria with respect to the desired goal. It is made in the form of pairwise comparisons using individual questionnaires. Since the evaluation criteria are subjective and qualitative in nature, it is very difficult for the decision maker to express the preferences using exact numerical values. That's why a special numerical scale [4] which consists of interpretation of linguistic terms is used (see Table 6).

Table 6. Saaty's pairwise comparison scale

| Numeric value | Linguistic terms |
|---------------|---|
| 1 | Equally important |
| 3 | Moderately more important |
| 5 | Strongly more important |
| 7 | Very Strongly more important |
| 9 | Extremely important |
| 2, 4, 6, 8 | The intermediate values that are used to address situations of uncertainty between the two adjacent judgments |

Results of comparisons of all experts are presented in the form of matrices (see Table 7).

Table 7. Criteria pairwise comparisons obtained by experts

| | Professional skills | | | Communicative skills | | | Personal qualities | | |
|-----------------|---------------------|--------|--------|----------------------|--------|--------|--------------------|--------|--------|
| | Exp. 1 | Exp. 2 | Exp. 3 | Exp. 1 | Exp. 2 | Exp. 3 | Exp. 1 | Exp. 2 | Exp. 3 |
| Prof. skills | 1 | 1 | 1 | 5 | 5 | 4 | 3 | 2 | 1 |
| Comm. skills | 1/5 | 1/5 | 1/4 | 1 | 1 | 1 | 1/3 | 1/4 | 1/5 |
| Pers. qualities | 1/3 | 1/2 | 1 | 3 | 4 | 5 | 1 | 1 | 1 |

Before calculating the weights, the consistency of the comparison matrix is checked. As a rule, only if consistency is less than 0.1, it is considered as acceptable, otherwise the pair-wise comparisons should be revised. In this decision making process, all of them are less than 0.092 that shows answers are consistent.

On the basis of Table 7 the final matrix is created by finding a mean between estimates of all experts (see Table 8). This metric is used because of solid decision to make all experts' voices to be

equal.

Table 8. Aggregate matrix with criteria pairwise comparisons

| | Professional skills | Communicative skills | Personal qualities |
|-----------------|---------------------|----------------------|--------------------|
| Prof. skills | 1 | 4,66 | 2 |
| Comm. skills | 0,22 | 1 | 0,25 |
| Pers. qualities | 0,6 | 4 | 1 |

The matrix from Table 8 is used in order to calculate criteria priority weights. The same way as it was earlier, a pairwise comparison of all the sub-criteria, with respect to each criterion, included in the decision-making model, is made. Obtained results are shown in Table 9.

Table 9. Criteria and subcriteria priority weights

| | | | |
|-------------------------|---------|-------------------------------------|---------|
| 1. Professional skills | 54,772% | 1.1. New task types creation | 25,232% |
| | | 1.2. HW checking | 26,068% |
| | | 1.3. Experience in LMS | 3,472% |
| 2. Communicative skills | 10,069% | 2.1. Openness to students | 2,946% |
| | | 2.2. Communication with teachers | 1,288% |
| | | 2.3. Participation in consultations | 5,834% |
| 3. Personal qualities | 35,159% | 3.1. Punctuality | 13,086% |
| | | 3.2. Ethical compliance | 10,062% |
| | | 3.3. Self-motivation | 12,011% |

Next step consists of deriving the relative priorities (preferences) of the alternatives with respect to each criterion. Overall priority weights of assistants are calculated by summing all local priorities. Final figures are shown in Table 10. Bar chart is built on the basis of overall preferences of the alternatives (see Fig. 4).

Table 10. Local and overall priorities of alternatives

| | A | B | C | D | E | F | G |
|--------------|---------|---------|---------|---------|---------|--------|--------|
| 1.1. | 8,352% | 6,711% | 3,784% | 2,728% | 1,821% | 1,214% | 0,622% |
| 1.2. | 3,140% | 6,060% | 9,131% | 3,836% | 1,700% | 0,966% | 1,234% |
| 1.3. | 0,160% | 0,249% | 0,148% | 1,001% | 1,618% | 0,148% | 0,148% |
| 2.1. | 0,203% | 0,154% | 0,101% | 0,441% | 0,082% | 0,803% | 1,161% |
| 2.2. | 0,444% | 0,089% | 0,117% | 0,062% | 0,062% | 0,444% | 0,072% |
| 2.3. | 0,414% | 0,387% | 0,198% | 0,833% | 0,144% | 1,628% | 2,230% |
| 3.1. | 1,035% | 2,677% | 1,921% | 3,670% | 1,663% | 1,041% | 1,078% |
| 3.2. | 0,254% | 1,928% | 0,254% | 3,822% | 1,782% | 0,993% | 1,029% |
| 3.3. | 0,923% | 2,040% | 0,761% | 4,201% | 1,640% | 1,108% | 1,339% |
| Total weight | 14,924% | 20,296% | 16,416% | 20,595% | 10,010% | 8,846% | 9,114% |

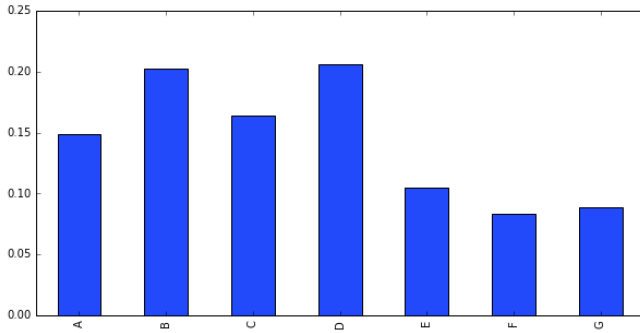


Figure 4. Overall preferences of the alternatives

VI. A DISCUSSION ON AHP RESULTS

AHP analysis shows that the prompt decision of choosing A as the best assistant is totally unfair. Results reveal that experts did not take into account other important criteria that in general overweighted those, which were chosen at first. Another discovered problem of A is some of his/her estimates, which are the worst in comparison with others (for instance, criteria 3.1 and 3.2). This fact also decreases his/her chances to be a leader.

The main interesting point of results are the highest figures which belong to both two assistants B and D. Let's describe each of them.

Assistant B can not be named as a brilliant employee. Nevertheless, he/she has showed good stable work without having bad results in any of the activities during the course. Despite not being the best in any of the criteria, B always was close to the leader.

In the same manner as B, assistant D has shown quite strong results in technical and communication estimates. In addition, D was on the top in the personal qualities criteria. He/she produces an impression of too self-motivated person and D was never late on any events. Result of D exceeds B at an inconspicuous gap of 0.3. Since experts make an arrangement on having no less than 2% advantage taking by the leader, such difference is admitted being not crucial for them.

In addition, there is a problematic situation with evaluation of the five best assistants. Four employees can be determined more or less clearly (A, B, C, and D). However, the difference between E and the closest competitor G is less than 1%, which is also insignificant.

VII. FUZZY TYPE-2 AHP

Since experts want to be more confident in fairness of their choice, we decide to apply another MCDM approach for purpose of aiming our goal. It is called Fuzzy AHP. In classical AHP crisp numbers are used, for pairwise comparison evaluations. However, in Fuzzy AHP, the linguistic variables are represented as fuzzy numbers instead of crisp. In this case a fuzzy logic provides a mathematical strength to capture the uncertainties associated with human cognitive process. Many researchers [5], [6] who have studied the Fuzzy AHP have provided evidence that it shows relatively more sufficient description of decision making processes compared to the traditional AHP methods.

According to [7], the membership functions of type-1 fuzzy sets have no uncertainty associated with it. Type-2 fuzzy sets

generalize type-1 fuzzy sets and systems so that more uncertainty for defining membership functions can be handled. That's why type-2 fuzzy logic is used.

A type-2 fuzzy set \tilde{A} in the universe of discourse X can be represented by a type-2 membership function $\mu_{\tilde{A}}$, shown as follows [8]:

$$\tilde{A} = \left\{ \left((x, u), \mu_{\tilde{A}}(x, u) \right) \mid \forall x \in X, \forall u \in J_x \subseteq [0, 1], 0 \leq \mu_{\tilde{A}}(x, u) \leq 1 \right\},$$

where J_x denotes an interval $[0, 1]$.

\tilde{A} is called an interval type-2 fuzzy set if all $\mu_{\tilde{A}} = 1$ [8]. Interval type-2 fuzzy sets are the most commonly used type-2 fuzzy sets because of their simplicity and reduced computational effort with respect to general type-2 fuzzy sets. For this reason, we use interval type-2 fuzzy sets.

A trapezoidal interval type-2 fuzzy set is illustrated as

$$\tilde{A}_i = (\tilde{A}_i^U, \tilde{A}_i^L) = \left(a_{i1}^U, a_{i2}^U, a_{i3}^U, a_{i4}^U; H_1(\tilde{A}_i^U), H_2(\tilde{A}_i^U), a_{i1}^L, a_{i2}^L, a_{i3}^L, a_{i4}^L; H_1(\tilde{A}_i^L), H_2(\tilde{A}_i^L) \right),$$

where \tilde{A}_i^U and \tilde{A}_i^L are type-1 fuzzy sets, $a_{i1}^U, a_{i2}^U, \dots, a_{i3}^U, a_{i4}^U$ are the reference points of the interval type-2 fuzzy set \tilde{A}_i ; $H_j(\tilde{A}_i^U)$ denotes the membership value of the element $a_{i(j+1)}^U$ in the upper trapezoidal membership function \tilde{A}_i^U and $H_j(\tilde{A}_i^L)$ denotes the membership value of the element $a_{i(j+1)}^L$ in the lower trapezoidal membership function \tilde{A}_i^L , $j = 1..2$ [7].

Pairwise comparison matrices got from experts for AHP are directly applied for our needs in Fuzzy AHP. We introduce interval trapezoidal type-2 fuzzy scales of the linguistic variables (see Table 11). They represent a modified version of scales proposed by [8] and include intermediate values between the two adjacent judgments like in AHP.

The priority weights of criteria (Table 12) and sub-criteria (Table 13) are demonstrated.

Type 2 fuzzy and defuzzified overall weights of the alternatives are shown in Tables 14 and 15. Bar chart is built on the basis of overall preferences of the alternatives (see Fig. 5). For defuzzification the Defuzzified Trapezoidal Type-2 Fuzzy Set (DTraT) approach is used proposed by [11].

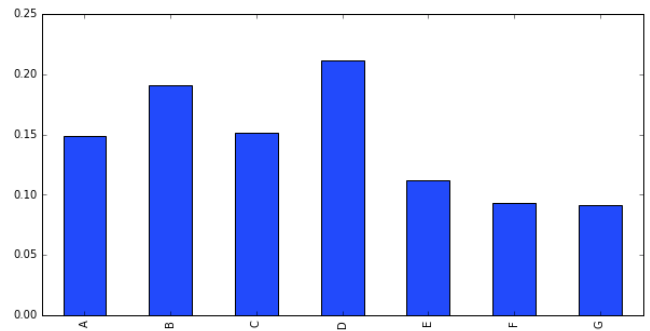


Figure 5. Overall preferences of the alternatives

VIII. A DISCUSSION ON FUZZY TYPE-2 AHP RESULTS

Now, we see that assistant D has higher priority weight than B and difference between them (2%) is suitable for experts. In addition, it can be noticed that E should be in the top five group, for sure (difference is also about 2%). Thus, Fuzzy AHP does not change ranks of alternatives but makes it clearer. It means that more reliable results are maintained since interval type-2 fuzzy

sets can better represent uncertainties.

Table 11. Trapezoidal interval type-2 fuzzy scales

| Numeric value from AHP | Trapezoidal interval type-2 fuzzy scales |
|------------------------|---|
| 1 | (1, 1, 1, 1; 1, 1) (1, 1, 1, 1; 1, 1) |
| 2 | (1, 1, 3, 4; 1, 1) (1.2, 1.2, 2.8, 3.8; 0.8, 0.8) |
| 3 | (1, 2, 4, 5; 1, 1) (1.2, 2.2, 3.8, 4.8; 0.8, 0.8) |
| 4 | (2, 3, 5, 6; 1, 1) (2.2, 3.2, 4.8, 5.8; 0.8, 0.8) |
| 5 | (3, 4, 6, 7; 1, 1) (3.2, 4.2, 5.8, 6.8; 0.8, 0.8) |
| 6 | (4, 5, 7, 8; 1, 1) (4.2, 5.2, 6.8, 7.8; 0.8, 0.8) |
| 7 | (5, 6, 8, 9; 1, 1) (5.2, 6.2, 7.8, 8.8; 0.8, 0.8) |
| 8 | (6, 7, 8.5, 9; 1, 1) (6.2, 7.2, 8.3, 8.8; 0.8, 0.8) |
| 9 | (7, 8, 9, 9; 1, 1) (7.2, 8.2, 8.8, 9; 0.8, 0.8) |

Table 12. Interval type-2 fuzzy weights of criteria

| Criteria | Interval type-2 weights |
|-------------------------|--|
| 1. Professional skills | (0.275, 0.377, 0.754, 1.005; 1, 1) (0.304, 0.410, 0.706, 0.935; 0.8, 0.8) |
| 2. Communicative skills | (0.057, 0.073, 0.14, 0.211; 1, 1) (0.061, 0.078, 0.13, 0.19; 0.8; 0.8) |
| 3. Personal qualities | (0.188, 0.257, 0.519, 0.708; 1, 1) (0.203, 0.274, 0.477, 0.639; 0.8, 0.8) |

Table 13. Interval type-2 fuzzy weights of sub-criteria

| Sub-criteria | Interval type-2 weights |
|--------------|--|
| 1.1. | (0.071, 0.134, 0.447, 0.811; 1, 1) (0.085, 0.154, 0.396, 0.703; 0.8, 0.8) |
| 1.2. | (0.074, 0.138, 0.453, 0.811; 1, 1) (0.088, 0.158, 0.402, 0.705; 0.8, 0.8) |
| 1.3. | (0.013, 0.022, 0.069, 0.124; 1, 1) (0.015, 0.025, 0.061, 0.108; 0.8, 0.8) |
| 2.1. | (0.008, 0.014, 0.062, 0.149; 1, 1) (0.009, 0.016, 0.052, 0.118; 0.8, 0.8) |
| 2.2. | (0.004, 0.007, 0.031, 0.071; 1, 1) (0.004, 0.008, 0.025, 0.055; 0.8, 0.8) |
| 2.3. | (0.014, 0.029, 0.115, 0.24; 1, 1) (0.017, 0.034, 0.099, 0.199; 0.8, 0.8) |
| 3.1. | (0.055, 0.087, 0.212, 0.317; 1, 1) (0.062, 0.095, 0.191, 0.28; 0.8, 0.8) |
| 3.2. | (0.046, 0.069, 0.168, 0.265; 1, 1) (0.051, 0.075, 0.151, 0.23; 0.8, 0.8) |
| 3.3. | (0.046, 0.075, 0.196, 0.317; 1, 1) (0.052, 0.082, 0.175, 0.274; 0.8, 0.8) |

IX. SENSITIVITY ANALYSIS

It is important to note that, contrary to the common belief, the system does not determine the decision we should make, rather, the results should be interpreted as a blueprint of preference and alternatives based on the level of importance obtained for the different criteria taking into consideration our comparative judgments. In other words, the AHP methodology allows us to determine which alternative is the most consistent with our criteria and the level of importance that we give them.

Taking this point into account, Sensitivity Analysis is used. It

performs a “what-if” analysis to see how the final results would have changed if the weights of the criteria would have been different [9].

Let’s start with a goal of finding the best teaching assistant. The first criterion has the highest weigh in our results ($\approx 50\%$). If we decrease its weight and proportionally increase other weights then D will still be a leader. In this case D will have even more clear-cut victory. Otherwise, if we increase weigh of this criterion up to 60% and more, then B will become a new leader. However, stakeholders come to one opinion that no one criterion should cost more than a half and they has highlited that the first criterion (professional skills) should stay as the most important one.

It means that weigh of the first criterion should be in the next approximate range [33%; 50%].

Let’s now tune proportions of the second and the third criteria. Calculations show that D can stop be a winner only and only if the third criterion will cost more than the second. Thus, this point was brought to expert discussion and they have unanimously decided that personal qualities (third criterion) should be appreciated higher than communication ones.

Another important note is change of proportions of subcriteria inside their criteria. There are no strong disputes about subcriteria weights, experts’ opinions differ no more than 10%. In this case change of subcriteria preferences in that range does not influence on the leader.

It means that there is no opportunity to have another leader than D by introducing small changes in current proportions of criteria weights.

At the same time, there is a complex situation with choosing top five assistants group. Analysis shows that four assistants are determined clearly. They are A, B, C, and D. The fifth assistant can be either E or G.

Calculations reveal that position of assistant G is directly connected with the second criteria and if its weight is equal or more than 15% than G will be in top five group instead of E. However, now second criterion has only nearly 10%.

Finally, after Sensitivity Analysis is done, next recommendations for the experts are given:

1. To choose assistant D as a winner.
2. To prolongate contracts with A, B, C and D.
3. To prolongate contract with E if experts think that personal qualities should be at least twice more important than communicating skills (finally, communicating skills should have a weight less than 15).
4. To prolongate contract with G, in other case.

X. FINAL RESULT AND CONCLUSIONS

Taking into consideration recommendations mentioned above, group of teachers has decided to follow first two instructions. They have selected D as the best teaching assistant on the course of “Discrete mathematics”. Also, they have prolonged contracts with D and A, B, and C assistants.

Table 14. Local and overall priorities of alternatives A, B, C, D

| | A | B | C | D |
|---------------------------------|--|--|--|--|
| 1.1. | (0.115, 0.212, 0.505, 0.806; 1, 1) (0.133, 0.234, 0.462, 0.723; 0.8, 0.8) | (0.098, 0.174, 0.414, 0.692; 1, 1) (0.113, 0.191, 0.378, 0.612; 0.8, 0.8) | (0.048, 0.091, 0.252, 0.484; 1, 1) (0.056, 0.101, 0.226, 0.413; 0.8, 0.8) | (0.037, 0.063, 0.173, 0.342; 1, 1) (0.042, 0.07, 0.154, 0.29; 0.8, 0.8) |
| 1.2. | (0.04, 0.067, 0.158, 0.272; 1, 1) (0.045, 0.073, 0.143, 0.238; 0.8, 0.8) | (0.083, 0.144, 0.314, 0.471; 1, 1) (0.095, 0.157, 0.291, 0.429; 0.8, 0.8) | (0.186, 0.265, 0.477, 0.657; 1, 1) (0.202, 0.283, 0.447, 0.608; 0.8, 0.8) | (0.067, 0.108, 0.26, 0.406; 1, 1) (0.077, 0.12, 0.239, 0.366; 0.8, 0.8) |
| 1.3. | (0.038, 0.042, 0.055, 0.067; 1, 1) (0.039, 0.043, 0.053, 0.064; 0.8, 0.8) | (0.038, 0.051, 0.082, 0.106; 1, 1) (0.041, 0.054, 0.078, 0.101; 0.8, 0.8) | (0.032, 0.037, 0.051, 0.067; 1, 1) (0.033, 0.038, 0.049, 0.063; 0.8, 0.8) | (0.177, 0.229, 0.382, 0.536; 1, 1) (0.187, 0.242, 0.361, 0.496; 0.8, 0.8) |
| 2.1. | (0.029, 0.044, 0.086, 0.133; 1, 1) (0.032, 0.047, 0.08, 0.12; 0.8, 0.8) | (0.027, 0.038, 0.072, 0.11; 1, 1) (0.029, 0.041, 0.067, 0.1; 0.8, 0.8) | (0.021, 0.026, 0.046, 0.071; 1, 1) (0.022, 0.028, 0.043, 0.064; 0.8, 0.8) | (0.064, 0.104, 0.237, 0.409; 1, 1) (0.072, 0.113, 0.217, 0.359; 0.8, 0.8) |
| 2.2. | (0.185, 0.262, 0.448, 0.587; 1, 1) (0.2, 0.278, 0.426, 0.554; 0.8, 0.8) | (0.041, 0.05, 0.077, 0.106; 1, 1) (0.043, 0.052, 0.073, 0.098; 0.8, 0.8) | (0.044, 0.055, 0.084, 0.111; 1, 1) (0.046, 0.057, 0.08, 0.104; 0.8, 0.8) | (0.038, 0.043, 0.061, 0.076; 1, 1) (0.039, 0.045, 0.058, 0.072; 0.8, 0.8) |
| 2.3. | (0.033, 0.046, 0.091, 0.137; 1, 1) (0.035, 0.05, 0.084, 0.124; 0.8, 0.8) | (0.034, 0.046, 0.083, 0.122; 1, 1) (0.036, 0.049, 0.078, 0.111; 0.8, 0.8) | (0.019, 0.025, 0.046, 0.07; 1, 1) (0.02, 0.026, 0.042, 0.063; 0.8, 0.8) | (0.065, 0.1, 0.22, 0.366; 1, 1) (0.072, 0.108, 0.201, 0.323; 0.8, 0.8) |
| 3.1. | (0.029, 0.042, 0.086, 0.13; 1, 1) (0.032, 0.045, 0.079, 0.116; 0.8, 0.8) | (0.102, 0.161, 0.324, 0.462; 1, 1) (0.113, 0.174, 0.302, 0.425; 0.8, 0.8) | (0.038, 0.052, 0.095, 0.138; 1, 1) (0.041, 0.055, 0.089, 0.126; 0.8, 0.8) | (0.134, 0.215, 0.483, 0.71; 1, 1) (0.153, 0.237, 0.446, 0.649; 0.8, 0.8) |
| 3.2. | (0.015, 0.02, 0.035, 0.053; 1, 1) (0.016, 0.021, 0.033, 0.048; 0.8, 0.8) | (0.068, 0.102, 0.198, 0.295; 1, 1) (0.075, 0.109, 0.185, 0.27; 0.8, 0.8) | (0.015, 0.02, 0.035, 0.053; 1, 1) (0.016, 0.021, 0.033, 0.048; 0.8, 0.8) | (0.163, 0.269, 0.543, 0.78; 1, 1) (0.184, 0.292, 0.505, 0.72; 0.8, 0.8) |
| 3.3. | (0.05, 0.059, 0.09, 0.115; 1, 1) (0.052, 0.061, 0.086, 0.108; 0.8, 0.8) | (0.061, 0.089, 0.23, 0.325; 1, 1) (0.07, 0.099, 0.209, 0.294; 0.8, 0.8) | (0.038, 0.047, 0.081, 0.115; 1, 1) (0.04, 0.049, 0.076, 0.105; 0.8, 0.8) | (0.184, 0.262, 0.492, 0.632; 1, 1) (0.203, 0.283, 0.463, 0.594; 0.8, 0.8) |
| Total fuzzy weight | (0.018, 0.052, 0.372, 1.069; 1, 1) (0.023, 0.064, 0.302, 0.826; 0.8, 0.8) | (0.026, 0.074, 0.496, 1.336; 1, 1) (0.034, 0.091, 0.407, 1.044; 0.8, 0.8) | (0.023, 0.061, 0.384, 1.062; 1, 1) (0.029, 0.074, 0.316, 0.827; 0.8, 0.8) | (0.035, 0.09, 0.552, 1.459; 1, 1) (0.045, 0.11, 0.454, 1.136; 0.8, 0.8) |
| Total defuzzy weight | 0.331 | 0.426 | 0.337 | 0.471 |
| Total normalized defuzzy weight | 14.868% | 19.119% | 15.124% | 21.128% |

Table 15. Local and overall priorities of alternatives E, F, G

| | E | F | G |
|---------------------------------|--|--|--|
| 1.1. | (0.025, 0.042, 0.123, 0.257; 1, 1) (0.029, 0.046, 0.109, 0.212; 0.8, 0.8) | (0.019, 0.03, 0.082, 0.161; 1, 1) (0.021, 0.033, 0.073, 0.134; 0.8, 0.8) | (0.013, 0.018, 0.04, 0.077; 1, 1) (0.014, 0.019, 0.036, 0.065; 0.8, 0.8) |
| 1.2. | (0.029, 0.046, 0.106, 0.192; 1, 1) (0.033, 0.05, 0.097, 0.165; 0.8, 0.8) | (0.021, 0.028, 0.055, 0.091; 1, 1) (0.022, 0.03, 0.05, 0.08; 0.8, 0.8) | (0.024, 0.033, 0.076, 0.131; 1, 1) (0.026, 0.036, 0.068, 0.111; 0.8, 0.8) |
| 1.3. | (0.283, 0.378, 0.56, 0.674; 1, 1) (0.302, 0.4, 0.532, 0.648; 0.8, 0.8) | (0.032, 0.037, 0.051, 0.067; 1, 1) (0.033, 0.038, 0.049, 0.063; 0.8, 0.8) | (0.032, 0.037, 0.051, 0.067; 1, 1) (0.033, 0.038, 0.049, 0.063; 0.8, 0.8) |
| 2.1. | (0.015, 0.02, 0.037, 0.06; 1, 1) (0.016, 0.021, 0.034, 0.053; 0.8, 0.8) | (0.128, 0.193, 0.392, 0.613; 1, 1) (0.14, 0.208, 0.363, 0.553; 0.8, 0.8) | (0.177, 0.282, 0.546, 0.772; 1, 1) (0.198, 0.304, 0.509, 0.715; 0.8, 0.8) |
| 2.2. | (0.038, 0.043, 0.061, 0.076; 1, 1) (0.039, 0.045, 0.058, 0.072; 0.8, 0.8) | (0.226, 0.292, 0.461, 0.589; 1, 1) (0.239, 0.306, 0.44, 0.559; 0.8, 0.8) | (0.042, 0.049, 0.068, 0.086; 1, 1) (0.044, 0.05, 0.066, 0.081; 0.8, 0.8) |
| 2.3. | (0.014, 0.017, 0.031, 0.049; 1, 1) (0.014, 0.018, 0.029, 0.044; 0.8, 0.8) | (0.135, 0.2, 0.409, 0.6; 1, 1) (0.148, 0.216, 0.377, 0.543; 0.8, 0.8) | (0.179, 0.272, 0.537, 0.747; 1, 1) (0.2, 0.295, 0.5, 0.693; 0.8, 0.8) |
| 3.1. | (0.064, 0.094, 0.194, 0.316; 1, 1) (0.07, 0.1, 0.179, 0.28; 0.8, 0.8) | (0.046, 0.064, 0.128, 0.209; 1, 1) (0.049, 0.068, 0.118, 0.184; 0.8, 0.8) | (0.05, 0.067, 0.13, 0.197; 1, 1) (0.053, 0.071, 0.12, 0.175; 0.8, 0.8) |
| 3.2. | (0.088, 0.146, 0.327, 0.538; 1, 1) (0.099, 0.159, 0.3, 0.48; 0.8, 0.8) | (0.047, 0.067, 0.145, 0.263; 1, 1) (0.051, 0.072, 0.132, 0.227; 0.8, 0.8) | (0.056, 0.077, 0.145, 0.225; 1, 1) (0.06, 0.082, 0.135, 0.203; 0.8, 0.8) |
| 3.3. | (0.111, 0.129, 0.195, 0.233; 1, 1) (0.117, 0.135, 0.186, 0.222; 0.8, 0.8) | (0.059, 0.074, 0.131, 0.175; 1, 1) (0.063, 0.078, 0.122, 0.159; 0.8, 0.8) | (0.064, 0.078, 0.137, 0.167; 1, 1) (0.067, 0.082, 0.127, 0.154; 0.8, 0.8) |
| Total weight | (0.021, 0.049, 0.283, 0.789; 1, 1) (0.026, 0.059, 0.233, 0.604; 0.8, 0.8) | (0.015, 0.035, 0.227, 0.681; 1, 1) (0.018, 0.042, 0.183, 0.509; 0.8, 0.8) | (0.016, 0.037, 0.232, 0.652; 1, 1) (0.019, 0.044, 0.188, 0.496; 0.8, 0.8) |
| Total defuzzy weight | 0.251 | 0.208 | 0.205 |
| Total normalized defuzzy weight | 11.347% | 9.334% | 9.079% |

The main important step now is to choose the fifth assistant. Before making a choice, experts decide to use a retrospective and to look through all methods that were applied earlier. Lecture of

the course noticed that since A, B, C, and D assistants are already confirmed it means that nobody will be responsible for commu-

nication with students (answering questions, having consultations) because assistant F did it before. However, now there is a choice between either E or G. And in this case G demonstrates a clear superiority compared with others as he/she is one of the top in this kind of work. Finally, G is chosen.

At the very beginning teachers wanted to choose assistant A as the best teaching assistant. However, the soft methods of analysis helped us to choose another assistant. Also, neither AHP nor Fuzzy AHP chose G teaching assistant as the 5th best assistant in the group. Only a sound logic helped us to do this.

The application of methods of system analysis can help to make a decision but it does not make a choice for us. We should look carefully at the results of system analysis methods, but make a balanced and considered decision.

BIBLIOGRAPHY

- [1] J. Lopa, "Using Undergraduate Students as Teaching Assistants," *The Professional & Organizational Development Network in Higher Education*, vol. 21, pp. 50-62, 2009.
- [2] M. Asad, S. Kermani and H. Hora, "A Proposed Framework for Evaluating Student's Performance and Selecting the Top Students in E-Learning System, Using Fuzzy AHP Method," in *Proceedings of the International Conference on Management, Economics and Humanities*, Istanbul-Turkey, 2015.
- [3] T. Saaty, "There is no mathematical validity for using fuzzy number crunching in the analytic hierarchy process," *Journal of Systems Science and Systems Engineering*, vol. 15, no. 4, pp. 457-464, 2006.
- [4] A. Monk and S. Howard, "The Rich Picture: A Tool for Reasoning About Work Context," [Online]. Available: <http://www-users.york.ac.uk/~am1/RichPicture.pdf>.
- [5] Changing Minds, "CATWOE," [Online]. Available: <http://creatingminds.org/tools/catwoe>.
- [6] M. Velasquez and P. Hester, "An Analysis of Multi-Criteria Decision Making Methods," vol. 10, no. 2, pp. 56-66, 2013.
- [7] T. Saaty, *The Analytic Hierarchy Process*, New York: McGraw Hill, 1980.
- [8] C. Boender and J. De Graan, "Multicriteria Decision Analysis with Fuzzy Pairwise Comparisons," vol. 29, pp. 133-143, 1989.
- [9] D. Chang, "Applications of The Extent Analysis Method on Fuzzy-AHP," vol. 95, pp. 649-655, 1996.
- [10] J. Mendel and R. John, "Type-2 fuzzy sets made simple," vol. 10, no. 2, pp. 117-127, 2002.
- [11] C. Kahraman and B. Öztayşi, "Fuzzy analytic hierarchy process with interval type-2 fuzzy sets," vol. 59, pp. 48-57, 2014.
- [12] E. Mu and M. Pereyra-Rojas, *An Introduction to the Analytic Hierarchy Process (AHP) Using Super Decisions*, vol. 2, New York: Springer, 2016.

Analysis of Mathematical Formulations of Capacitated Vehicle Routing Problem and Methods for their Solution

Ekaterina Beresneva

Faculty of Computer Science
National Research University Higher School of Economics
Moscow, Russia
eberesneva@hse.ru

Scientific Advisor: Prof. Sergey Avdoshin

Software Engineering School
National Research University Higher School of Economics
Moscow, Russia
savgdoshin@hse.ru

Abstract— Vehicle Routing Problem (VRP) is one of the most widely known questions in a class of combinatorial optimisation problems. It is concerned with the optimal design of routes to be used by a fleet of vehicles to serve a set of customers. In this study we analyse Capacitated Vehicle Routing Problem (CVRP) – a subcase of VRP, where the vehicles have a limited capacity. CVRP is aimed at savings in the global transportation costs. Typical applications of CVRP are delivery of goods, solid waste collection, street cleaning etc. The problem is NP-hard, therefore heuristic algorithms which provide near-optimal polynomial-time solutions will be considered instead of the exact ones. The aim of this article is to present a new adaptation of mathematical formulations of CVRP and to make a survey on methods for solving each type of this problem. This paper provides an overview of the most perspective methods to be realized in later works.

Keywords—survey, classification, mathematical formulations, capacitated vehicle routing problem.

I. INTRODUCTION

The Vehicle Routing Problem (VRP) is one of the most widely known questions in a class of combinatorial optimization problems. VRP is directly related to Logistics transportation problem and it is meant to be a generalization of the Travelling Salesman Problem (TSP). In contrast to TSP, VRP produces solutions containing some (usually, more than one) looped cycles, which are started and finished at the same point called “depo”. The objective is to minimize the cost (time or distance) for all tours. For the identical type of input data, VRP has higher solving complexity than TSP. Both problems belong to the class of NP-hard tasks. Specialised algorithms are able to consistently find optimal solutions for cases with up to about 50 customers; larger problems have been solved to optimality in some cases, but often at the expense of considerable computing time. Thus, actuality of research and development of heuristics algorithms for solving VRP is on its top, because such approximate algorithms can produce near-optimal solutions in a polynomial time. It is especially important in real-based tasks when there are more than one hundred clients in a delivery net.

Real world applications may be mail delivery, solid waste collection, street cleaning, distribution of commodities, design

telecommunication, transportation networks, school bus routing, dial-a-ride systems, transportation of handicapped persons, and routing of sales people and maintenance units. A survey of real-world applications is in [1].

This work is aimed at analysis of VRP subcase, which is called Capacitated Vehicle Routing Problem (Capacitated VRP, CVRP), where the vehicles have a limited capacity. It means that there is a physical restriction on transportation more than determined amount of weight for each machine. Capacitated vehicle routing problems CVRP form the core of logistics planning and are hence of great practical and theoretical interest.

Unfortunately, there are no articles concerned with CVRP, which have both a full classification of the subcases and description of the solving algorithms. In addition, all observed math models are based on linear programming. The purpose of this study is to present a new adaptation of mathematical formulations of CVRP and to make a survey on heuristic methods for solving each extension of this problem.

Clearly, a study of this type is inevitably restricted by various constraints, in this research only CVRP subcases with static and deterministic input are considered instead of the dynamic and stochastic ones. Another condition is that classification is based according to various types of constraints.

The paper is structured as follows. In the second part, the classical mathematical formulations of CVRP are described. After that, in the third section, some notes on solution methods of the problems are provided. In the fourth part, a classification of most popular subcases of CVRP is given, including description of additional constraints with their math formulations. This section also includes most perspective methods that can be applied for solving special types of CVRP. Finally, the fifth part consists of scheme with basic problems of the CVRP class and their interconnections and of conclusion.

II. CLASSICAL CVRP

In this paper, mathematical formulation of Asymmetrical CVRP (ACVRP) proposed by [2] is adopted in a new way as follows. ACVRP is chosen for basic formulation instead of

Symmetrical CVRP (SCVRP) because the first one is a general variant of the second problem. In the paper we will use CVRP abbreviation having in mind the next formulation.

Given a complete weighted oriented graph $G = (V, A)$. Let $I = \{0, 1, \dots, N\}$, where $N = |V|$. Graph vertices are indexed as $ind = V \rightarrow I, (\forall v \in V)(\forall w \in V)(v \neq w \Rightarrow ind(v) \neq ind(w))$. Thus, $V = \{v_0, v_1, \dots, v_N\}$ is set of vertices, here $i = ind(v_i)$, and A is set of arcs. Let v_0 be a depot, where vehicles are located, and v_i be the destination points of a delivery, $i \neq 0$.

The distance between two vertices v_i and v_j is calculated using a distance function $c(v_i, v_j)$. Here a real-valued function $c: V \times V \rightarrow \mathbb{R}$ satisfies [3]:

1. $c(v_i, v_j) \geq 0$ (*non-negativity axiom*).
2. $c(v_i, v_j) = 0$ if and only if $v_i = v_j$ (*identity axiom*).

Each destination vertex $v_i, i = \overline{0..N}$, is associated with a known nonnegative demand, $d(v_i)$, to be delivered, and the depot has a fictitious demand $d(v_0) = 0$. The total demand of the set $V' \subseteq V$ is calculated as follows: $d(V') = \sum_{v_i \in V'} d(v_i)$.

Let $K = const$ be a number of available vehicles at the depot v_0 . Each vehicle has the same capacity – C . Let us assume that every vehicle may perform at most one and $K \geq K_{min}$, where K_{min} is a minimal number of vehicles needed to serve all the customers due to restriction on C . Clearly, next condition must be fulfilled – $(\forall v_i \in V)(d(v_i) \leq C)$, which prohibits goods transportation that exceed maximum vehicle capacity.

Let introduce $V^0 = \{v_0\}$, where $v_0 \in V$. We divide V in $K + 1$ sets: $S = \{V^0, V^1, \dots, V^K\}$, each subset, except for V^0 , represent a set of customers to be served for one vehicle. S^{all} is a set of all possible partitions of V . Let $J = \{0, 1, \dots, K\}$ be a set that keeps indexes. Then $(\forall i \in J)(|V^i| \geq 1)$. There should be no duplicates in any of subsets from S^{all} : $(\forall S \in S^{all})(\forall i \in J)(\forall j \in J)(i \neq j \Rightarrow V^i \cap V^j = \emptyset)$. Also, all subsets from S must form set V . Thus, $V = \bigcup_{i=0}^K V^i$. In this notation, we should make $V^{0i} = V^0 \cup V^i, i = \overline{1..K}$. It is obvious that $d(V^{0i}) \leq C, i = \overline{1..K}$.

Let introduce $M^i = \{1, \dots, N^i\}, N^i = |V^i|, \sum_{i=1}^K N^i = N$. Then $M^{0i} = \{0\} \cup M^i$. Let $I^i = \{ind(v) | v \in V^i\}$ be a set of vertex indices from V^i . Then $I^{0i} = \{0\} \cup I^i$.

Let $H^i = \{p^i: M^{0i} \rightarrow I^{0i} | p^i(0) = 0 \text{ \& } (\forall x \in M^{0i})(\forall y \in M^{0i})(x \neq y \Rightarrow p^i(x) \neq p^i(y))\}$ be a set of codes p^i of all Hamiltonian cycles $h^i = (v_{p^i(0)}, v_{p^i(1)}, \dots, v_{p^i(N^i)})$ of V^{0i} .

Weight of a Hamiltonian cycle $h^i \in H^i$ can be found using p^i according to the formula:

$$f(p^i) = c(v_{p^i(0)}, v_{p^i(N^i)}) + \sum_{j=0}^{N^i-1} c(v_{p^i(j)}, v_{p^i(j+1)})$$

Let S' be a set of $\{V^{01}, V^{02}, \dots, V^{0K}\}$. It should be noticed that a set of Hamiltonian cycles h^i depends on S , thus, in this

notation the weight of S' is calculated as $F(S', p^1, \dots, p^K) = \sum_{i=\overline{1..K}} f(p^i)$.

Overall, the formulation of CVRP is to find:

$$(S_0, p_0^1, \dots, p_0^K) = \arg \min_{(S, p^1, \dots, p^K) \in S^{all} \times H^1 \times \dots \times H^K} F(S, p^1, \dots, p^K)$$

If $c(v_i, v_j) = c(v_j, v_i)$ for $\forall v_i \in V \forall v_j \in V$ then the problem is symmetrical (SCVRP) and triangle inequality axiom must be hold $c(v_i, v_k) \leq c(v_i, v_j) + c(v_j, v_k)$.

According to [1], another variant of mathematical formulation of CVRP allows to leave some vehicles unused, it means that at most K circuits must be determined. Of course, the number of K_{min} must be less or equal than K .

In this case the basic formulation described above should be changed as follows:

We subsequently divide V in $K' + 1$ sets: $S = \{V^0, V^1, \dots, V^{K'}\}$, where $K' = \overline{K_{min}..K}$. And all K from basic formulation should be replaced by K' .

Alternative variant takes its place from real-based situations where available vehicles have their own capacity $C_i, i = \overline{1..K}$. Due to this fact, next restriction appears:

$$d(V^{0i}) \leq C_i, i = \overline{1..K}.$$

However, most researches put this alternative to another class of problems not connected with CVRP which is known as the Mixed Fleet VRP or as the Heterogeneous Fleet VRP. Thus, this variant will not be taken into consideration in this paper.

Among the best-known heuristic algorithms are those proposed by Pisinger and Ropke (2007) [4], Nagata and Braysy (2009) [5], and Vidal et al. (2012) [6].

III. SOLUTION METHODS

Before classification it should be noted, that there are three types of algorithms that are used to solve any subcase VRP:

- *Exact algorithms*. They find an optimal solution but take a great time for solving of large instances. Such methods include branch and bound, cutting plane, branch and cut, column generation, cut and solve, branch-and-cut-and-price, branch-and-price, and dynamic programming.

- *Classical heuristics*. They look for a quite good solution without guarantee of optimality. In comparison to exact methods, they work faster. Such methods include constructive heuristics, two phase methods, improvement heuristics.

- *Metaheuristics*. Such type of algorithms is also called a framework for building heuristics. According to [7], metaheuristics are classified into three groups: metaheuristics based on local search, on population (natural inspired), and hybrid metaheuristics.

IV. EXTENSIONS OF CVRP

A. Open VRP (OVRP)

The OVRP is a variant of the CVRP where the vehicles need not return to the depot after visiting the last customer of a given route. Any OVRP instance can be converted to an ACVRP instance by simply setting $c(v_i, v_0) = 0$.

There is only one heuristic algorithm for solving OVRP proposed by Salari et al. (2010) [8]. Their method is based on Integer Linear Programming Improvement Procedure.

There is a good variety of metaheuristics. Most known and important are following algorithms: Hybrid evolution strategy algorithm by Repoussis et al. (2010) [9], variant of Variable Neighborhood Search (VNS) algorithm for OVRP by Fleszar et al. (2009) [10], method based on Tabu Search (TS) with route-evaluations memories by Zachariadis and Kiranoudis (2010) [11], Yu et al. (2011) Genetic algorithm and the last one is Particle swarm optimization metaheuristic proposed by MirHassani and Abolghasemi (2011) [12].

B. Distance-Constrained CVRP (DCVRP) [13]

The next extension of CVRP to be considered is Distance-Constrained CVRP. It suggests introducing the maximum length or time constraint for each route. It means that the total travelled distance by each vehicle in the solution is less than or equal to the maximum possible travelled distance f_{max} . And the next restriction must be hold: $(\forall i = 1..K)(f^T(h^i) \leq f_{max})$.

Most heuristics applied to simple CVRP can be easily converted for solving DCVRP cases. However, one heuristic proposed by Li et al. stands out from them [14]. It transforms the DCVRP into a multiple traveling salesman problem with time windows.

C. VRP with Time Windows (VRPTW)

In VRPTW there is a constraint on time interval $[a_i; b_i]$ associated with each v_i , called time window. It means that service of each customer must start only after the time a_i comes and this service must end before the time b_i . Obviously, $a_0 = 0$ and $b_0 = \infty$ for v_0 . Let us assume that if t_{cur} is a current time, then all vehicles leave v_0 when $t_{cur} = 0$. If a vehicle arrives to v_i at the moment when $t_{cur} < a_i$, then it is obliged to wait until $t_{cur} = a_i$ and to start serving only after that moment.

New function $t(v_i, v_j)$, returning travel time between v_i and v_j , appears. Also, a variable $sr v_i$ keeping serving time of v_i is introduced. It is clear, that the problem can be solved if $(\forall v_i \in V)(\exists v_j \in V)(a_i + sr v_i + t(v_i, v_j) + sr v_j \leq b_j)$.

There are a lot of metaheuristics for solving VRPTW, but the most actual and state-of-the-art ones are given. The guided Evolutionary algorithm of Repoussis et al. (2009) [15] combines evolution, ruin-and-recreate mutations and guided local search. Prescott-Gagnon et al. (2009) [16] suggests a Large Neighborhood search (LNS) combined with branch-and-price for solution reconstruction. The method proposed by

Nagata et al. (2010) [17] uses an interesting relaxation scheme with penalized returns in time. Another algorithm (Vidal et al. (2013)) [18] also applies time-constraint relaxations during the search to benefit from infeasible solutions in the search space.

D. VRP with Backhauls (VRPB)

VRPB is another extension to CVRP. To define VRPB we need to divide the set of customers V^i into two subsets: the first set contains customers who require the product to be delivered, these customers are called linehaul customers $L^i \subset V^i$. The other set contains customers who require the product to be picked up, they are called backhaul customers $B^i \subset V^i$. $L^i \cap B^i = \emptyset, L^i \cup B^i = V^i$. Also, neither all deliveries nor all pick-ups should exceed vehicle capacity: $d(L^i) \leq C$ & $d(B^i) \leq C$. If the tour contains customers from both sets, the linehaul customers must serve before any backhaul customers. Note that tours with backhaul customers only are not allowed in some formulations [1].

In basic formulation H^i should be changed as follows:

$$H^i = \{p^i: M^{oi} \rightarrow I^{oi} \mid p^i(0) = 0 \text{ \& } (\forall x \in M^{oi})(\forall y \in M^{oi}) \\ (x \neq y \Rightarrow p^i(x) \neq p^i(y)) \text{ \& } ((x < y) \Rightarrow ((v_{p^i(x)} \in L^i) \text{ or } \\ (v_{p^i(y)} \in B^i)))\}.$$

The best metaheuristics, according to [19], include the Adaptive LNS (ALNS) of Ropke and Pisinger (2006) [20], the Tabu Search (TS) of Zachariadis and Kiranoudis (2012) [21] which uses long-term memories to direct the search toward inadequately exploited characteristics; and finally multi-ant colony system algorithm by Gajpal and Abad (2009) [22], which suggests two multi-route local search schemes.

E. VRP with Backhauls and Time Windows (VRPBTW)

Like in VRPB, VRPBTW suggests having linehaul and backhaul customers. In addition, with every location v_i there is a service time $sr v_i$ associated for loading/unloading and a time window $[a_i; b_i]$, which specifies the time in which this service has to be provided. In the same way as for VRPTW, when arriving too early at a location v_i , i.e., before a_i , the vehicle is allowed to wait until a_i to start the service. Also, the linehaul customers must be served before any backhaul customers. Thus, mathematical formulation of VRPBTW is a combination of both formulations of VRPTW and VRP.

The most powerful algorithms for solving VRPBTW are those which are proposed by Thangiah et al. (1996) [23] and by Kucukoglu et al (2015) [24]. The first method is based on insertion procedure with improving through the application of λ -interchange and 2-opt exchange procedures. The second one includes combination of TS and SA.

F. VRP with Pickup and Delivery (VRPPD)

In the basic version of VRPPD, each customer v_i requests two demands: $d(v_i)$ to be delivered and $p(v_i)$ to be picked up. In addition, we need to add for each customer v_i two new

variables: $O(v_i)$ which denotes the vertex where the source of delivery originates and $D(v_i)$ which denotes the customer where the destination of the pick up exists. It should be noted that for each customer the delivery must be implemented before the pick up.

Let define $d(V_{part}^{oi}) - p(V_{part}^{oi}) \leq C, part = \overline{0..N^i}$, as the weight of the current load of the vehicle after visiting $v_{p^i(part)}$, where $d(V_{part}^{oi}) = \sum_{j \in [0..part]} d(v_j), part \leq N^i$ and $p(V_{part}^{oi}) = \sum_{j \in [0..part]} p(v_j)$.

In basic formulation H^i should be changed as follows:

$$H^i = \{p^i: M^{oi} \rightarrow I^{oi} \mid p^i(0) = 0 \& (\forall x \in M^{oi})(\forall y \in M^{oi}) \\ (x \neq y \Rightarrow p^i(x) \neq p^i(y)) \& (\forall x \in M^{oi})(\forall y \in M^{oi}) \\ ((v_{p^i(y)} = D(v_{p^i(x)})) \Rightarrow (x < y)) \& (\forall x \in M^{oi}) \\ (\forall y \in M^{oi})((v_{p^i(x)} = O(v_{p^i(y)})) \Rightarrow (x > y))\}$$

A great number of heuristics and metaheuristics are presented in [25].

G. VRP with Simultaneous Pickup and Delivery (VRSPD)

VRSPD is a subcase of VRPPD where each customer is a linehaul as well as a backhaul customer. In VRSPD each customer not only requires a given quantity of products to be delivered but also requires a given quantity of products to be picked up. A complete service (both delivery and pickup) to the customer is provided by a vehicle in a single visit. Thus, there is no need to explicitly indicate both variables $O(v_i)$ and $D(v_i)$ as in VRPPD.

It is found in the literature that the heuristics of Subramanian et al. (2010) [25], Zachariadis and Kiranoudis (2011) [26] and Souza et al. (2010) [27] together produce the best known results.

H. VRP with Mixed Pickup and Delivery (VRMPD)

VRMPD is also a subcase of VRPPD where each customer has either a delivery demand or pickup. Therefore, there is $d(v_i) > 0$ and $p(v_i) = 0$ in the first case and $p(v_i) > 0$ and $d(v_i) = 0$. Nevertheless, in basic formulation H^i should be changed the same way as it was shown for VRPPD.

The best known heuristics are those of Subramanian (2013) which is based on Iterative Local Search (ILS) idea [28] and hybrid algorithm proposed by Subramanian, Uchoa and Ochi (2013) [29].

I. VRP with Pickup and Delivery and Time Windows (VRPPDTW)

The VRPPDTW in this paper contains all constraints in the VRPPD plus added constraints in which both pickup and delivery have given time windows. With every location v_i there is a service time srv_i associated for loading/unloading and a time window $[a_i, b_i]$, which specifies the time in which this service has to be provided. In the same way as for VRPTW, when arriving too early at a location v_i , i.e., before a_i , the

vehicle is allowed to wait until a_i to start the service. Also, for each customer the delivery must be implemented before the pick up.

Efficient neighborhood-centered metaheuristics have been proposed, including the ALNS of Ropke and Pisinger (2006) [30] and the two-phase method of Bent, and Van Hentenryck (2006) [31], which combines SA to reduce the number of routes with LNS to optimize the distance. However, these methods were recently outperformed by the memetic algorithm of Nagata and Kobayashi (2011) [32], which exploits a well-designed crossover focused on transmitting parent characteristics without introducing too many new arcs in the offspring.

J. Multi-depot VRP (MDVRP)

The MDVRP is a generalization of the CVRP where more than one depot may be considered. Also, the vehicle must start and end at the same depot.

So, part of basic formulation should be changed as follows:

Let G be a number of depots. Let introduce $V^0 = \{v_0^1, v_0^2, \dots, v_0^G\}$, where $v_0^i \in V$. In this case, we should make $V^{oi} = \{v_0^j \in V^0\} \cup V^i, i = \overline{1..K}, j = \overline{1..G}$.

The best heuristic approaches for the MDVRP are considered to be developed by Pisinger and Ropke (2006) [33] and Vidal et al. (2012) [6].

K. VRP with Multiple Use of Vehicles (VRPM) or Multi-Trip VRP (MTVRP)

VRPM or MTVRP is a variant of standart CVRP in which the same vehicle can be assigned to several routes during a given planning period. Not only this constraint is introduced but also the sum of the durations of the trips assigned to the same vehicle must not exceed T_{max} . T_{max} is a trip duration being the sum of the travel times on arcs used in the route. Thus, new function $t(v_i, v_j)$, returning travel time between v_i and v_j , appears. Function $t: V \times V \rightarrow R$ satisfies the same axioms as c .

In this variant it is possible if $d(V^{oi}) > C, i = \overline{1..K}$. We additionally divide V^i in MT_i sets: $V^i = \{V_1^i, V_2^i, \dots, V_{MT_i}^i\}$, where $MT_i \in [1, |V^i|]$. Let $J = \{0, 1, \dots, K\}$ be a set that keeps indexes. Then $(\forall i \in J) (\forall mt \in \overline{1..MT_i}) (|V_{mt}^i| \geq 1)$. There should be no duplicates in any of subsets from V^i : $(\forall i \in J) (\forall mt_1 \in \overline{1..MT_i}) (\forall mt_2 \in \overline{1..MT_i}) (mt_1 \neq mt_2 \Rightarrow V_{mt_1}^i \cap V_{mt_2}^i = \emptyset)$. Also, $V^i = \bigcup_{mt=1}^{MT_i} V_{mt}^i$. In this notation, we should make $(\forall mt \in \overline{1..MT_i}) (V_{mt}^{oi} = V^0 \cup V_{mt}^i)$. It is obvious that $(\forall mt \in \overline{1..MT_i}) (d(V_{mt}^{oi}) \leq C)$.

Let introduce $M_{mt}^i = \{1, \dots, N_{mt}^i\}, N_{mt}^i = |V_{mt}^i|, \sum_{mt=1}^{MT_i} N_{mt}^i = N^i$. Then $M_{mt}^{oi} = \{0\} \cup M_{mt}^i$. Let $I_{mt}^i = \{i \mid i = ind(v), v \in V_{mt}^i\}$ be a set of vertex indices from V_{mt}^i . Then $I_{mt}^{oi} = \{0\} \cup I_{mt}^i$.

Let $H_{mt}^i = \{p_{mt}^i: M_{mt}^{0i} \rightarrow I_{mt}^{0i} | p_{mt}^i(0) = 0 \text{ \& } (\forall x \in M_{mt}^{0i})(\forall y \in M_{mt}^{0i})(x \neq y \Rightarrow p_{mt}^i(x) \neq p_{mt}^i(y))\}$ be a set of codes of all Hamiltonian cycles $h_{mt}^i = (v_{p_{mt}^i(0)}, v_{p_{mt}^i(1)}, \dots, v_{p_{mt}^i(N_{mt}^i)})$ of V_{mt}^{0i} .

Weight of a Hamiltonian cycle $h_{mt}^i \in H_{mt}^i$ can be found according to the formula:

$$f(h_{mt}^i) = c(v_{p_{mt}^i(0)}, v_{p_{mt}^i(N_{mt}^i)}) + \sum_{j=0}^{N_{mt}^i-1} c(v_{p_{mt}^i(j)}, v_{p_{mt}^i(j+1)})$$

Let S' be a set of $\{V_{mt}^{01}, V_{mt}^{02}, \dots, V_{mt}^{0K}\}$, $mt = \overline{1..MT_i}$. In this notation the weight of S' is calculated as $F(S') = \sum_{i=\overline{1..K}} \sum_{mt=\overline{1..MT_i}} f(h_{mt}^i)$.

Weight of a Hamiltonian cycle $h_{mt}^i \in H^i$ can be found using p_{mt}^i as $f^T(h_{mt}^i) = t(v_{p_{mt}^i(0)}, v_{p_{mt}^i(N_{mt}^i)}) + \sum_{j=0}^{N_{mt}^i-1} t(v_{p_{mt}^i(j)}, v_{p_{mt}^i(j+1)})$. The most important thing here is the next constraint: $(\forall i = \overline{1..K})(\sum_{mt=1}^{MT_i} f^T(h_{mt}^i) \leq T_{Max})$.

Overall, the formulation of VRPM is to find:

$$\begin{aligned} S^0, h_{mt}^1, \dots, h_{mt}^K: F(S^0, h_{mt}^1, \dots, h_{mt}^K) \\ = \min_{S \in S^{all}, mt=\overline{1..MT_i}} F(S, h_{mt}^1, \dots, h_{mt}^K) \end{aligned}$$

Metaheuristic inspired by ideas of TS and adaptive memory-based search (AMS) (Taillard (1993) [34]) still shows good results. In addition, another variant of AMS by Olivera and Viera (2007) [35] is considered to be competitive.

L. Periodic VRP (PVRP)

The Periodic VRP (PVRP) is used when planning is done over a certain period and deliveries to the customer can be done in different days. For the PVRP, customers can be visited more than once, though often with limited frequency.

Efficient algorithm for solving PVRP is parallel extension of UTS with neighborhood-centered search (Cordeau and Maischberger, 2012 [36]). Also, the VNS of Hemmelmayr et al. (2009) [37], and the hybrid record-to-record and integer programming matheuristic of Gulczynski et al. (2011) [38] can be successfully applied. And one more metaheuristic is one that proposed by Vidal et al. (2012) [6]. It produces the current best solutions by combining the GA search breadth with efficient LS, relaxations schemes, and diversity management procedures.

M. Split Delivery VRP (SDVRP)

In the SDVRP-MDA, more than one vehicle can service a customer, so that a customer's demand can be split among several vehicles on different routes. The most important here is that split deliveries are allowed only if at least a minimum fraction of a customer's demand is delivered by each vehicle visiting the customer.

The first matheuristic for a SDVRP is proposed in Chen et al. (2007) [39]. The idea of the approach is based on

combination of the classical Clarke and Wright algorithm, the Mixed-Integer Linear Programming (MILP) model and variable length record-to-record travel methods. A similar procedure is applied in Gulczynski et al. (2010) [40] to the SDVRP with minimum delivery amounts, that is a SDVRP where each delivery to a customer should consist of at least a minimum amount of goods. Another metaheuristic which contains TS approach is proposed in 2008 by Archetti et al. [41] The main thing here is to obtain a reduced graph by removing some arcs and to apply a set covering MILP formulation for the best routes. And in Jin et al. (2008) [42] a set covering formulation is proposed and the problem is solved through column generation.

N. Cumulative CVRP (CCVRP)

CCVRP minimizes the sum of the arrival times at the customers instead of minimizing the total distance (or travel time) as an objective.

For the CCVRP, Ngueveu et al. (2010) [43] and Ribeiro and Laporte (2012) [44] modified the hybrid GA. Also, two-phase metaheuristic proposed by Ke and Feng in 2013 [45] is considered to be successful enough.

V. CONCLUSION

The presented study is undertaken in order to make a survey on CVRP subcases and on heuristic methods for solving each extension of this problem. In addition, author variants of mathematical formulations for some subcases of CVRP are given. It should be noticed that a great work is done because more than 80 sources (articles, books, theses etc) were analyzed and intercompared. Not all are listed in references due to page limits and different contribution to the paper.

Fig. 1. The basic problems of the CVRP class and their interconnections.

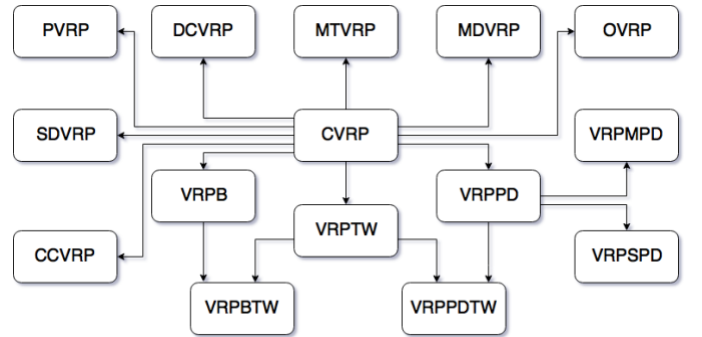


Figure 1 sums up relations between classes of the CVRP and forms the classification of its subtypes.

In our future work, we are going to extend current survey by adding dynamic and stochastic subcases of CVRP. Also, due to the fact that in this paper some extensions have not got full description of math models, it will be provided soon.

REFERENCES

- [1] P. Toth and D. Vigo, "An overview of vehicle routing problems," in *The Vehicle Routing Problem*, SIAM, 2002.

- [2] G. B. Dantzig and J. H. Ramser, "The Truck Dispatching Problem," *Management Science*, vol. 6, no. 1, pp. 80-91, 1959.
- [3] M. Reed and B. Simon, *Methods of Modern Mathematical Physics*, London: Academic Press, 1972.
- [4] P. Pisinger and S. Ropke, "A general heuristic for vehicle routing problems," *Computers & Operations Research*, vol. 34, no. 8, pp. 2403-2435, 2007.
- [5] Y. Nagata and O. Braysy, "Edge assembly-based memetic algorithm for the capacitated vehicle routing problem," *Networks*, vol. 54, no. 4, pp. 205-215, 2009.
- [6] T. Vidal, T. Crainic, M. Gendreau, N. Lahrichi and W. Rei, "A hybrid genetic algorithm for multi-depot and periodic vehicle routing problems," *Operations Research*, vol. 60, no. 3, pp. 611-624, 2012.
- [7] B. Golden, S. Raghavan and E. Wasil, *The vehicle routing problem: Latest advances and new challenges*, New York: Springer, 2008.
- [8] M. Salari, P. Toth and A. Tramontani, "An ILP improvement procedure for the Open Vehicle Routing Problem," *Computers & Operations Research*, vol. 37, no. 12, pp. 2106-2120, 2010.
- [9] P. Repoussis, C. Tarantilis, O. Braysy and G. Ioannou, "A hybrid evolution strategy for the open vehicle routing problem," *Computers & Operations Research*, vol. 37, no. 3, pp. 443-455, 2010.
- [10] K. Fleszar, I. Osman and K. Hindi, "A variable neighbourhood search algorithm for the open vehicle routing problem," *European Journal of Operational Research*, vol. 195, no. 3, pp. 803-809, 2009.
- [11] E. Zachariadis and C. Kiranoudis, "An open vehicle routing problem metaheuristic for examining wide solution neighborhoods," *Computers & Operations Research*, vol. 37, no. 4, pp. 712-723, 2010.
- [12] S. MirHassani and N. Abolghasemi, "A particle swarm optimization algorithm for open vehicle routing problem," *Expert Systems with Applications*, vol. 38, no. 9, pp. 11547-11551, 2011.
- [13] G. Laporte, Y. Nobert and M. Desrochers, "Optimal routing under capacity and distance restrictions," *Operations Research*, vol. 33, no. 5, p. 1050-1073, 1985.
- [14] C. Li, D. Simchi-Levi and M. Desrochers, "On the distance constrained vehicle routing problem," *Operational Research*, vol. 40, pp. 790-799, 1992.
- [15] P. Repoussis, C. Tarantilis and G. Ioannou, "Arc-guided evolutionary algorithm for the vehicle routing problem with time windows," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 3, p. 624-647, 2009.
- [16] E. Prescott-Gagnon, G. Desaulniers and L. Rousseau, "A branch-and-price-based large neighborhood search algorithm for the vehicle routing problem with time windows," *Networks*, vol. 54, no. 4, p. 190-204, 2009.
- [17] Y. Nagata, O. Bräysy and W. Dullaert, "A penalty-based edge assembly memetic algorithm for the vehicle routing problem with time windows," *Computers & Operations Research*, vol. 37, no. 4, p. 724-737, 2010.
- [18] T. Vidal, T. Crainic, M. Gendreau and C. Prins, "A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows," *Computers & Operations Research*, vol. 40, no. 1, p. 475-489, 2013.
- [19] K. Braekers, K. Ramaekers and I. Nieuwenhuysse, "The vehicle routing problem: State of the art classification and review," *Computers & Industrial Engineering*, vol. 99, pp. 300-313, 2016.
- [20] S. Ropke and D. Pisinger, "A unified heuristic for a large class of vehicle routing problems with backhauls," *European Journal of Operational Research*, vol. 171, no. 3, p. 750-775, 2006.
- [21] E. Zachariadis and C. Kiranoudis, "An effective local search approach for the vehicle routing problem with backhauls," *Expert Systems with Applications*, vol. 39, no. 3, p. 3174-3184, 2012.
- [22] Y. Gajpal and P. Abad, "Multi-ant colony system (MACS) for a vehicle routing problem with backhauls," *European Journal of Operational Research*, vol. 196, no. 1, p. 102-117, 2009.
- [23] S. Thangiah, J.-Y. Potvin and T. Sun, "Approaches to Vehicle Routing with Backhauls and Time," *Windows International Journal of Computers and Operations Research*, vol. 23, no. 11, pp. 1043-1057, 1996.
- [24] I. Küçükoğlu and N. Öztürk, "An advanced hybrid meta-heuristic algorithm for the vehicle routing problem with backhauls and time windows," *Computers and Industrial Engineering*, vol. 86, no. 3, pp. 60-68, 2015.
- [25] A. Subramanian, Drummond, C. Bentes, L. Ochi and R. Farias, "A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery," *Computers & Operations Research*, vol. 37, no. 11, pp. 1899-1911, 2010.
- [26] E. Zachariadis and C. Kiranoudis, "A local search metaheuristic algorithm for the vehicle routing problem with simultaneous pick-ups and deliveries," *Expert Systems with Applications*, vol. 38, no. 3, pp. 2717-2726, 2011.
- [27] M. Souza, M. Silva, M. Mine, L. Ochi and A. Subramanian, "A hybrid heuristic, based on iterated local search and GENIUS, for the vehicle routing problem with simultaneous pickup and delivery," *International Journal of Logistics Systems Management*, vol. 10, no. 2, pp. 142-156, 2010.
- [28] A. Subramanian and M. Battarra, "An iterated local search algorithm for the travelling salesman problem with pickups and deliveries," *Journal of the Operational Research Society*, vol. 64, pp. 402-409, 2013.
- [29] A. Subramanian, E. Uchoa and L. Ochi, "A hybrid algorithm for a class of vehicle routing problems," *Computers & Operations Research*, vol. 40, no. 10, pp. 2519-2531, 2013.
- [30] S. Ropke and D. Pisinger, "An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows," *Transportation Science*, vol. 40, no. 4, p. 455-472, 2006.
- [31] R. Bent and P. Van Hentenryck, "A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows," *Computers & Operations Research*, vol. 33, no. 4, p. 875-893, 2006.
- [32] Y. Nagata and S. Kobayashi, "A memetic algorithm for the pickup and delivery problem with time windows using selective route exchange crossover," *Proceedings of PPSN'11*, vol. 6238, p. 536-545, 2011.
- [33] D. Pisinger and S. Ropke, "A general heuristic for vehicle routing problems," *Computers & Operations Research*, vol. 34, no. 8, p. 2403-2435, 2007.
- [34] E. Taillard, G. Laporte and M. Gendreau, "Vehicle routing with multiple use of vehicles," *Journal of the Operational Research Society*, vol. 47, no. 8, p. 1065, 1996.
- [35] A. Olivera and O. Viera, "Adaptive memory programming for the vehicle routing problem with multiple trips," *Computers & Operations Research*, vol. 34, no. 1, p. 28-47, 2007.
- [36] J.-F. Cordeau and M. Maischberger, "A Parallel Iterated Tabu Search Heuristic for Vehicle Routing Problems," *Computers & Operations Research*, vol. 39, no. 9, p. 2033-2050, 2012.
- [37] V. Hemmelmayr, K. Doerner and R. Hartl, "A variable neighborhood search heuristic for periodic routing problems," *European Journal of Operational Research*, vol. 195, no. 3, p. 791-802, 2009.
- [38] D. Gulczynski, B. Golden and E. Wasil, "The period vehicle routing problem : New heuristics and real-world variants," *Transportation Research Part E : Logistics and Transportation Review*, vol. 47, no. 5, pp. 648-668, 2011.
- [39] S. Chen, B. Golden and E. Wasil, "The split delivery vehicle routing problem: Applications, algorithms, test problems, and computational results," *Networks*, vol. 49, p. 318-329, 2007.
- [40] D. Gulczynski, B. Golden and E. Wasil, "The split delivery vehicle routing problem with minimum delivery amounts," *Transportation Research Part E*, vol. 46, p. 612-626, 2010.
- [41] C. Archetti and M. Speranza, "The split delivery vehicle routing problem: a survey," in *The Vehicle Routing Problem Latest Advances and New Challenges*, *Operations Research*, Computer Science Interfaces Series, 2008, pp. 103-122.
- [42] M. Jin, K. Liu and B. Eksioğlu, "A column generation approach for the split delivery vehicle routing problem," *Operations Research Letters*, vol. 36, pp. 265-270, 2008.
- [43] S. Ngueveu, C. Prins and C. Wolfler, "An effective memetic algorithm for the cumulative capacitated vehicle routing problem," *Computers & Operations Research*, vol. 37, no. 11, p. 1877-1885, 2010.
- [44] G. Ribeiro and G. Laporte, "An adaptive large neighborhood search heuristic for the cumulative capacitated vehicle routing problem," *Computers & Operations Research*, vol. 39, no. 3, p. 728-735, 2012.
- [45] L. Ke and Z. Feng, "A two-phase metaheuristic for the cumulative capacitated vehicle routing problem," *Computers & Operations Research*, vol. 40, no. 2, pp. 633-638, 2013.

Auto-calibration and synchronization of camera and MEMS-sensors

Alexander Polyakov
Saint Petersburg
State University
polyakov.alx@gmail.com

Anastasiya Kornilova
Saint Petersburg
State University
kornilova.anastasiia@gmail.com

Iakov Kirilenko
Saint Petersburg
State University
y.kirilenko@spbu.ru

Abstract—In this paper, we present a robust auto calibration and synchronization algorithm for a system which consists of camera and MEMS-sensors (gyroscopes). The main task of our research is to find such parameters as the focal length of camera and time offset between sensor timestamps and frame timestamps which is caused by frame processing and encoding. This auto calibration makes possible to scale computer vision algorithms (video stabilization, 3D reconstruction, video compression), which use frames and sensor data, to a wider range of devices with camera and MEMS-sensors. In this article, we present a review and comparison of current approaches to calibration and propose our improvements for these methods which increase the quality of previous works and applicable for a general model of video stabilization algorithm with MEMS-sensors.

I. INTRODUCTION

The high quality of frames, received from modern smartphone cameras, expands the frontiers of solutions in computer vision tasks. Lately, there are more and more attempts to scale current practices in such areas of computer vision as video stabilization [1], [2], [3], [4], augmented reality[5], 3D reconstruction [6], [7], photogrammetry on mobile platforms and embedded systems. But these algorithms demand big computational resources that not allows to apply them to above-mentioned platforms and, moreover, in real time.

The presence of numerous different sensors on these platforms, caused by the low cost of their production and high precision at the same time, allows using their data effectively. As the majority of above-stated tasks is any way connected with detection of camera movement (which is the “bottleneck” in most algorithms), the main preference is given to motion sensors – gyroscope and accelerometer [8], [9].

Expansion of mathematical model of computer vision algorithm not only increases quality and reduces calculations, but gives rise to new difficulties. In particular, besides general intrinsic parameters of the camera (focal length, optical center, rolling shutter) there are parameters of sensors (i.e, bias for gyroscope) and parameters of model “camera-sensors” (camera and sensors orientation, camera and sensors synchronization parameters). Therefore, if desired to scale an algorithm to a large amount of platforms (for example, in case of mobile phones) automatic calibration of these parameters is needed. It is caused by a big variety of cameras, sensors and their combinations.

This work is a continuation of the research [10] conducted on a subject of real-time digital video stabilization using MEMS-sensors and aims to prototype and implement an algorithm of auto-calibration of key parameters for this task: focal length and parameters of synchronization of frames and gyroscope data.

II. PRELIMINARIES

This section is devoted to basic definitions, general mathematical models, and agreements which will come out throughout this work.

A. Pinhole camera model

Pinhole camera model is a basic mathematical camera model which describes a mapping from 3-dimensional real world to its projection onto the image. This mapping satisfies the formula, in which X is coordinates of a point in real world and x is coordinates of its projection. Also, it depends on camera parameters: f – focal length, (o_x, o_y) – optical center [11].¹

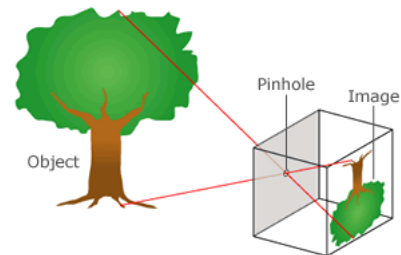


Figure 1. Pinhole camera model

$$\begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{pmatrix} f_x & 0 & -o_x \\ 0 & f_y & -o_y \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

B. Rotation camera model

In case of camera rotation in space using rotation operator R , we get the next relationship between two projections x_1 and x_2 of one point in space X caught at a different time t_1 (rotation R_1) and t_2 (rotation R_2) correspondingly.

¹Image is taken from the website https://en.wikipedia.org/wiki/Pinhole_camera_model

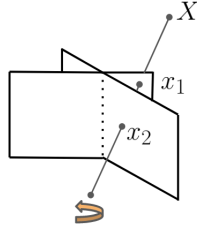


Figure 2. Rotation camera model

$$x_1 = KR(t_1)X$$

$$x_2 = KR(t_2)X$$

By transforming these expressions, the following needed relationship is established:

$$x_2 = KR(t_2)R^T(t_1)K^{-1}x_1$$

Thus, the matrix of image transformation between moments in time t_1 and t_2 is defined as:

$$W(t_1, t_2) = KR(t_2)R^T(t_1)K^{-1}$$

$$x_2 = W(t_1, t_2)x_1$$

C. Rolling shutter effect

"Rolling shutter" is an effect arising on the majority of CMOS cameras, at which each row of the frame is shot at different time due to vertical shutter.²



Figure 3. Object movement

When shutter scans the scene vertically, the moment in time at which each point of the frame is shot, directly depends on the row it is located in. Thus, if i is the number of the frame and y is the row of that frame, then the moment at which it was shot can be calculated this way:

$$t(i, y) = t_i + t_s \frac{y}{h}$$

where t_i is the moment when frame number i was shot, t_s is the time it takes to shot a single frame, h is the height of the frame. This can be used to make the general model more precise, when calculating the image transformation matrix.

²Images are taken from the website <http://www.red.com/learn/red-101/global-rolling-shutter>

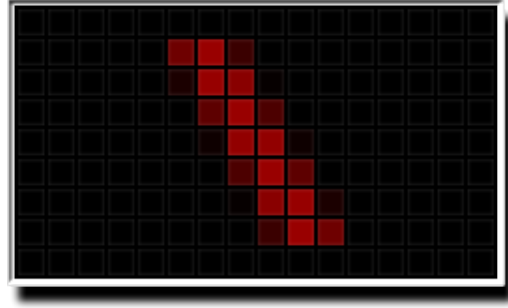


Figure 4. Rolling-shutter effect during capturing the moving object

D. Gyroscope

The gyroscope is a sensor (MEMS-sensor in our case) which sends information about angular velocities of a body. Using this data and its timestamps, a rotation matrix (rotation operator) can be calculated through integration.

There are two approaches for integration data of gyroscope with different computational complexity and accuracy. The first approach is linear integration for receiving Euler angles and then their transformation to a rotation matrix, where θ – is rotation angle of one axis and ω – velocity over this axis between t and $t + \delta$:

$$\theta(t + \delta) = \theta(t) + \int_t^{t+\delta} \omega(t)dt, \quad (1)$$

This approach is applied only in case of insignificant and small rotations, because of the imperfection of Euler angles as an algebraic structure. The other and more complex approach is to use quaternions for data integration. This article [12] gives a full description about the integration of angular velocities using quaternions, and we tend to apply it.

E. Stabilization quality metrics

There are two main metrics which can estimate the quality of video stabilization of static scene – RMSE (root mean square error) and ITF (inter-frame transformation fidelity).

The first is a comparison between two frames pixel-by-pixel using typical L2 metric.

The ITF metric directly depends on PSNR (peak signal-to-noise ratio) parameter between two consecutive frames ($k, k + 1$):

$$PSNR(k) = 10 \log_{10} \frac{I_{max}}{MSE(k)}$$

where I_{max} is maximum pixel intensity, and is counted as:

$$ITF = \frac{1}{N-1} \sum_{k=1}^{N-1} PSNR(k)$$

where N is count of frames in the video.

F. Features

In the computer vision, feature is a pattern that satisfies certain properties and can be detected on the image. One of directions of feature use is feature matching which is mainly focused on searching of similar objects on two frames. In our work, we use feature matching to estimate how the camera moved through shooting.

In our experiments we have used two features types – ORB (Oriented FAST and rotated BRIEF) [13] and SIFT (Scale-Invariant Feature Transform) [14] which prove themselves as the most stable and robust in feature matching. SIFT is considered to exhibit the highest matching accuracies, but requires significant computational resources, while ORB is very fast but less precise [15].

G. Description of stabilization algorithm

At the moment stabilization algorithm, proposed in our previous paper[10], works as follows.

- 1) Integrate gyroscope data (angular velocities and timestamps) using quaternions.
- 2) Determine frame timestamp and corresponding rotation matrix.
- 3) Count transformation camera matrix for every horizontal section of the frame (typically, there are several gyro reading per frame and, consequently, several rotation matrices).
- 4) Transform every section using transformation matrix and combine them.
- 5) Write transformed frame to the video.

The algorithm stabilizes video like a tripod, at now complex camera motion is not supported, but in progress.

III. DETAILED PROBLEM DESCRIPTION

As it was mentioned in the description of the stabilization algorithm, it directly depends on camera parameters: focal length, optical center and rolling shutter parameter. In most cases, all parameters besides focal length can be got from API of the device on which this algorithm runs (at the moment the major advantage is given to Android platforms). Thus, one of the main goals of this research is to find focal length which is the most accurate for our stabilization algorithm.

The other significant direction is to synchronize frames received from the camera and data received from sensors. Mistiming is caused by the time needed for frame processing – scanning and encoding. Therefore, we need to find time offset of this processing to consider it in our model.

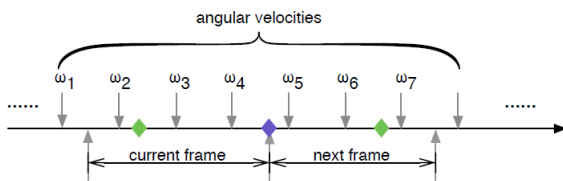


Figure 5. Matching the time series of frames and gyroscope

Thus, the main goal of this research is to find the suitable focal length and time offset. Some of the described methods are wider and cover other parameters, and we also consider this information.

IV. CALIBRATION ALGORITHMS

In this section we describe various approaches that we have tested during this research. The section contains a description of our basic method, review and implementation of the most known methods of calibration from other areas, and our improvements on these methods for our specific task.

A. Calibration based on stabilization metrics

focal length, time offset, rolling shutter

This simple approach is based on stabilization metrics described in section 2. Using ITF metric, we can estimate the quality of video stabilization after transformation of frames: the higher the value of metric – the better video is stabilized.

The approach determines three parameters: focal length, time offset and rolling shutter parameter and is as follows: detect a range and step of each parameter (for example, range of focal length – [500, ..., 1200] and step – 50) and find tuple of parameters on which metric is maximized using brute-force search.

It is worth noting, despite of the huge computational complexity this method gives the most accurate results due to the strong dependence on the current mathematical model.

B. OpenCV calibration method

focal length, optical center, distortion coefficients

This algorithm is applicable only in case of known geometry of subject which is on the scene. Also, the subject should contain easily distinguished feature points. This subject is usually called calibration pattern. We have used the main calibration pattern which is supported by OpenCV – chessboard. It depends on such parameters as size of chessboard, the distance between cells and others.

The algorithm also determines distortion coefficients³ and is as follows:

- 1) Count initial intrinsic parameters of the camera. Initial distortion coefficients are equal to zero.
- 2) Estimate camera position using this initial parameters using PnP method.
- 3) Using Levenberg-Marquardt algorithm minimize re-projection error – sum of square root distances between two matched point.

C. Grid search method

focal length, time offset

Using frames and gyroscope data we can estimate the motion of camera in two ways:

- 1) Use feature points on frames and estimate motion using the difference between matched points on consequence frames.

³https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

- 2) Use data of gyroscope – measurements and their timestamps.

This approach is as follows. Firstly, we determine two functions which describe the average measure of camera motions in two ways – using feature points and using gyroscope measurements. These functions must depend on time and if necessary must have facilities for interpolation (data of gyroscope is discrete). Having these functions, that describes motion in different ways, we can estimate shift (time offset) of functions using cross-correlation.

Let's determine these functions:

$$r_f(t) = \frac{\sum_{m \in M(t)} (m_x - m'_x) + (m_y - m'_y)}{2|M(t)|(t_i - t_{i-1})}$$

$$r_g(t) = \frac{\omega_x(t) + \omega_y(t) + \omega_z(t)}{3}$$

On the picture, you can see similar shape of these functions.

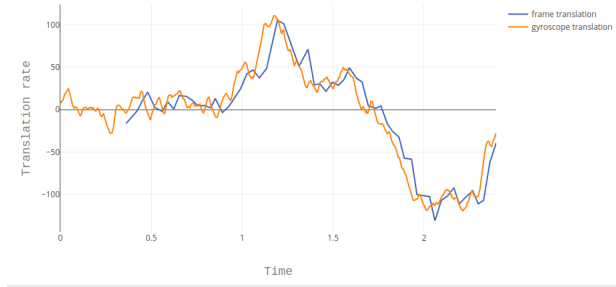


Figure 6. Time offset between frame and gyroscope

We have tried two typical cross-correlation functions to find offset:

$$s(a, b) = a * b$$

$$s(a, b) = -|a - b|$$

If we have a set of possible offsets T_d , we can find offset with a maximum value of correlation between frames and gyroscope functions:

$$offset = \arg \max_{t_d \in T_d} \sum_{t \in T} s(r_f(t - t_d), r_g(t))$$

Authors who support this approach tend to opinion that initial scale constant is a focal length value and try to find this constant like:

$$r_g(t) = f * r_f(t)$$

Using a method of the least squares:

$$f = \arg \min_f \sum_{i=1}^n (r_f(t_i + t_d) - f * r_g(t_i))^2$$

D. Improvements for grid search method

This method presents a combination of two methods – method with stabilization metrics and method with grid search. The time offset is found by grid search method. If we have a set of possible focal lengths F and the calculated value of time offset, we can calculate a value of focal length which maximizes stabilization metric.

$$f = \arg \max_{f \in F} ITF(f, t_d)$$

This method is suitable very well in case of using these time offset and focal length in our video stabilization algorithm.

Also, we have abandoned to take in account motion over z-axis, which is perpendicular to the camera matrix. This motion has non-linear correlation with linear angular velocity over this axis and leads to an error in the algorithm.

V. RESULTS OF PROTOTYPING

In this section we will describe results of experiments and conditions in which they were conducted.

A. Dataset and environment

Our algorithm was tested on a dataset which consists of video and gyroscope data from smartphones with the Android operating system. For these purposes, we have a special Android application which records mp4 video file and csv format file with stamps for gyroscope and frame events. This application supports mobile platforms starting with 21 level Android API because of in this API event-driven scheme for camera frames was supported by camera2 interface. The csv file consists of two types of strings: "f" – for frames and "X, Y, Z, timestamp" – for gyroscope readings.

A framework for calibration algorithm comparison was implemented in Python using OpenCV 3.4 library. It consists of modules for video and gyroscope file parsing and a module for integration of gyroscope readings using quaternion. The framework also has opportunities for calculating metric statics for every method.

We have tested our algorithms on a dataset from the smartphone with the following parameters:

- Model number: Xiaomi Redmi 3S;
- Android version: 6.0.1 (build MMB29M).

B. Experiments

Inside our framework, we have implemented all described algorithms and compare them using stabilization quality metrics. We have tested algorithms on different scene types and with different camera movements. An algorithm with stabilization metric was considered as standard. All results are presented in tables. We compare grid search method using different cross-correlation functions and different feature detectors.

Experiments show that OpenCV algorithm has the worst result because of it is very sensitive for the scene (user needs to use chessboard or other pattern) and rotation and is not fit for our mathematical model.

In the table, you can see results of grid search algorithm without/with improvements (metric) in comparison with stabilization metric algorithm.

The algorithm is parametrized with feature types and shows the best results with the second cross-correlation function (similarity function).

| Algorithm | Offset, μs | f | Metric |
|-----------------------------|-----------------|-----|--------|
| Metric (standard) | 45 | 850 | 14.04 |
| Grid Search + ORB | 45 | 825 | 13.97 |
| Grid Search + SIFT | 45 | 950 | 13.33 |
| Grid Search + Metric + ORB | 45 | 850 | 14.04 |
| Grid Search + Metric + SIFT | 45 | 850 | 14.04 |

| Algorithm | Offset, μs | f | Metric |
|-----------------------------|-----------------|-----|--------|
| Metric (standard) | 45 | 850 | 16.25 |
| Grid Search + ORB | 50 | 850 | 16.10 |
| Grid Search + SIFT | 40 | 925 | 15.87 |
| Grid Search + Metric + ORB | 50 | 850 | 16.10 |
| Grid Search + Metric + SIFT | 40 | 850 | 15.53 |

| Algorithm | Offset, μs | f | Metric |
|-----------------------------|-----------------|-----|--------|
| Metric (standard) | 45 | 850 | 15.82 |
| Grid Search + ORB | 45 | 950 | 15.05 |
| Grid Search + SIFT | 50 | 825 | 15.31 |
| Grid Search + Metric + ORB | 45 | 850 | 15.82 |
| Grid Search + Metric + SIFT | 50 | 850 | 15.30 |

The first two tables show the result of calibration in case of 1-dimensional motion. It is demonstrated that in case of ORB and SIFT features results are identical in accuracy. Also, results show that in case of metric improvements focal length after calibration is equal to standard in comparison with simple grid search.

The third table describes results of calibration in case of 2-dimensional motion. Results are equal to the case of 1-dimensional motion. As we discussed earlier, the algorithm does not consider 3-dimensional motion because of constraints of grid search model.

C. Main results

To sum up, experiments have demonstrated that:

- 1) grid search method shows the better result for our mathematical model of camera and camera motion;
- 2) using grid search method, the best calibration result is achieved with the second cross-correlation function (similarity function);
- 3) ORB and SIFT features show equals results in search of the time offset, therefore we can use ORB as a faster method of feature matching;
- 4) our improvements of grid search with stabilization metric allow to find focal length which is equal to standard;
- 5) the algorithm supports only two-dimensional motion (except motion over, axis which is perpendicular to camera matrix), but this is not a strong restriction for users, therefore, our algorithm can be used on a large scale.

VI. CONCLUSION

As lately cameras and motion sensors (gyroscope, accelerometer) very often tend to occur on one platform (smart-phones or embedded systems), the quantity of the algorithms, using their joint information, has significantly increased. These algorithms directly depends on parameters of the system "camera-sensors," such as focal length, rolling shutter, synchronization parameters, which differ from platform to platform, and therefore these parameters must be calibrated for increasing of scalability.

Our work proposes the method for auto-calibration of focal length and time series offset (synchronization parameter), which is the most suitable for our video stabilization algorithm using MEMS-sensors. We have review different approaches and choose the nearest for our specific task. We have found parameters for this method which increase the quality of the calibration algorithm.

It worth noting that proposed algorithm can be scaled not only for stabilization video task. It can be scaled for all algorithms which support our mathematical model of camera and camera movement.

In the future, we plan to expand the count of calibration parameters with rolling shutter parameter and parameter of relative orientation of the camera and sensor axes.

ACKNOWLEDGMENT

Funding for this work was provided by JetBrains Research.

References

- [1] S. Liu, M. Li, S. Zhu, and B. Zeng, "Codingflow: Enable video coding for video stabilization," vol. 26, pp. 1–1, Apr. 2017.
- [2] M. Grundmann, V. Kwatra, and I. Essa, *Auto-Directed Video Stabilization with Robust L1 Optimal Camera Paths*. 2011.
- [3] W.-C. Hu, C.-H. Chen, Y.-J. Su, and T.-H. Chang, "Feature-based real-time video stabilization for vehicle video recorder system," *Multimedia Tools and Applications*, vol. 77, no. 5, pp. 5107–5127, Mar. 1, 2018, issn: 1573-7721. doi: 10.1007/s11042-017-4369-7. [Online]. Available: <https://doi.org/10.1007/s11042-017-4369-7>.
- [4] F. Liu, M. Gleicher, J. Wang, H. Jin, and A. Agarwala, *Subspace video stabilization*. 2011.
- [5] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui, "Mobile augmented reality survey: From where we are to where we go," *IEEE Access*, vol. 5, pp. 6917–6950, 2017, issn: 2169-3536. doi: 10.1109/ACCESS.2017.2698164.
- [6] A. Bethencourt and L. Jaulin, "3d reconstruction using interval methods on the kinect device coupled with an imu," *International Journal of Advanced Robotic Systems*, vol. 10, no. 2, p. 93, 2013. doi: 10.5772/54656.

- [7] J. Rambach, A. Pagani, S. Lampe, R. Reiser, M. Pancholi, and D. Stricker, "Fusion of unsynchronized optical tracker and inertial sensor in ekf framework for in-car augmented reality delay reduction," in *2017 IEEE International Symposium on Mixed and Augmented Reality (ISMAR-Adjunct)*, Oct. 2017, pp. 109–114. doi: 10.1109/ISMAR-Adjunct.2017.43.
- [8] A. Karpenko, D. Jacobs, and J. Baek, *Digital Video Stabilization and Rolling Shutter Correction using Gyroscopes*. 2011.
- [9] S. Bell, A. Troccoli, and K. Pulli, *A Non-Linear Filter for Gyroscope-Based Video Stabilization*. 2014.
- [10] N. Z. A.V. Kornilova I.A. Kirilenko, "Real-time digital video stabilization using mems-sensors," *Proceedings of the Institute for System Programming*, vol. 29, no. 4, pp. 2220–6426, 2017, issn: 2169-3536. doi: 10.15514/ISPRAS-2017-29(4)-5.
- [11] R. Szeliski, *Computer Vision: Algorithms and Applications*. 2010.
- [12] J. Diebel, *Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors*. 2006.
- [13] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International Conference on Computer Vision*, Nov. 2011, pp. 2564–2571. doi: 10.1109/ICCV.2011.6126544.
- [14] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004, issn: 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. [Online]. Available: <https://doi.org/10.1023/B:VISI.0000029664.99615.94>.
- [15] E. Karami, S. Prasad, and M. S. Shehata, "Image matching using sift, surf, BRIEF and ORB: performance comparison for distorted images," *CoRR*, vol. abs/1710.02726, 2017. arXiv: 1710.02726. [Online]. Available: <http://arxiv.org/abs/1710.02726>.

An approach to a software implementation of architectural generative design for BIM

Mikhail Prokofyev

Chair of Computer-aided design
BMSTU Kaluga Branch

Kaluga Branch of Bauman Moscow State Technical
University, 2 Bazhenova Str., Kaluga, 248000, Russian
Federation
mp.prokofyev@gmail.com

Vladimir Kirillov

Chair of Computer-aided design
BMSTU Kaluga Branch

Kaluga Branch of Bauman Moscow State Technical
University, 2 Bazhenova Str., Kaluga, 248000, Russian
Federation
kvu@bmstu-kaluga.ru

Abstract—This paper describes a flexible workflow for generative design applied to architectural space planning. Generative design can be described as a process of exploration of design space according to high-level goals, constraints, and preferences defined by the user. Building software system that implements generative design methods is a complicated task. Existing state-of-art papers on architectural generative design sidestep an issue of problems emerging while developing and implementing generative methods into real solutions. Current research provides some systematization and generalization on generative architectural design and problems behind the implementation of generative evolutionary concepts on different stages. Awareness about system level problems could help developers to build highly efficient generative design software and components for new generation of BIM solutions.

Keywords—generative; design; architecture; space; planning; genetics; evolution; bim

I. INTRODUCTION

BIM or Building Information Modelling is a process for creating and managing information on a construction project across the project lifecycle. It brings together all of the information about every component of a building, in one place. Now BIM is in transition to the third “level of maturity” when questions of collaborative work, cloud computing, and open data formats arise and become urgent. In such conditions, generative design becomes a present-day topic of interest again. Applied to the conceptual design stage of building lifecycle it could make a dramatic impact on entire architectural practice. According to survey results on integration of BIM and generative design [1], about 80% of practitioners said that integration of BIM and generative design could help to overcome difficulties they face with on conceptual design stage.

II. GENERATIVE DESIGN

Generative design (GD) is a field, which incorporates architecture, design, and computations. It is a process of exploration of design space according to high-level goals,

constraints, and preferences defined by the user. After architectural space is observed, GD program reports to the designer which options it considers promising for further analysis. The goal of the process is to decrease the time of concept design development and provide stakeholders with a wider variety of design solutions.

On conceptual design stage, generative design system provides the designer with a much deeper exploration of complex design spaces. Potentially it helps to deliver more efficient complex design concepts to stakeholders much faster.

Mathematically speaking it is possible to describe generative design task as design optimization problem (1):

$$\min_x \mathbf{F}(x) \quad (1)$$

subject to $g_j(x) \leq 0; j = 1, 2, \dots, m; h_l(x) = 0; l = 1, 2, \dots, b;$
where (2)

$$\mathbf{F}(x) = \begin{bmatrix} F_1(x) \\ F_2(x) \\ \dots \\ F_k(x) \end{bmatrix} \quad (2)$$

is a vector of objective functions, k is the number of objective functions, m is the number of inequality constraints, and b is the number of equality constraints. $x = [x_1, x_2, \dots, x_n]$ is a vector of design variables.

Methods of objectives minimization can vary.

Marler and Arora [2] provide a comprehensive survey of optimization methods for engineering tasks. Research [3] contains a brief overview of other works related to generative design problem as well. Generative design approach proposed in this paper incorporates evolutionary (genetic) systems and adaptation paradigms into the architectural design process to solve the minimization problem.

For multiple-objective problems, the objectives are generally conflicting, preventing simultaneous optimization of

each objective. Genetic algorithms (GA) are a meta-heuristic that is particularly well suited for this class of problems. Traditional GA are customized to accommodate multi-objective problems by using specialized fitness functions and introducing methods to promote solution diversity [4].

The concept of GA was developed by Holland and his colleagues in the 1960s and 1970s [5]. GAs are inspired by the evolutionist theory explaining the origin of species.

In GA terminology, a solution vector $x \in X$ is called an individual or a chromosome. Chromosomes are made of discrete units called genes. Each gene controls one or more features of the chromosome. GA operate with a collection of chromosomes, called a population. GA use two operators to generate new solutions from existing ones: crossover and mutation. The crossover operator is the most important operator of GA. In the crossover, generally, two chromosomes, called parents, are combined together to form new chromosomes, called offspring. The mutation operator introduces random changes into characteristics of chromosomes. Mutation is generally applied at the gene level. Reproduction involves selection of chromosomes for the next generation. In the most general case, the fitness of an individual determines the probability of its survival for the next generation [4].

Another of the critical components in evolutionary design system is the relationship between genotype and phenotype. Using generative design notion phenotype is a high-level design representation while genotype is genetic chromosome code of the design. The generative approach described in this paper works on both of the levels.

III. PROBLEMS OF GENERATIVE DESIGN SYSTEMS

Existing state-of-art papers on architectural generative design, such as [3] or [6], sidestep an issue of problems emerging while developing and implementing generative methods into real software solutions. This knowledge about system level problems could help developers to build more efficient generative design software and components for new generation of BIM solutions.

A. Encoding problem

Normally, a chromosome corresponds to a unique solution x in the solution space. This requires a mapping mechanism between the solution space and the chromosomes. This mapping is called an encoding [4].

In fact, genetic evolution algorithms work on the encoding of a problem, not on the problem itself. Therefore, choosing encoding schema for generative design method is an essential task.

In the original implementation of GA by Holland, genes are assumed to be binary digits [4]. In later implementations, more varied gene types have been introduced: permutation encoding, floating point encoding, tree encoding, etc. Nevertheless, none of these encoding schemas are applicable to generative design without any modification. It means that it is very complicated to develop universal encoding method for

any type of design. Architectural conceptual planning task proposes that design space contains elements of different types and structure: walls, doors, windows, floor, etc.; and developing encoding schema we should decide what parts of the design are important for genetic representation.

Encoding schema depends on parametrization concepts and influences crossover and mutation operators logic.

B. Crossover and mutation problem

As it was mentioned earlier, crossover and mutation operators represent core evolution mechanisms and operate on the genotype level where encoding is important.

If parent chromosomes encode designs with different configurations, for example, with twelve and eleven rooms or zones or with different rooms placement, crossover operator should process such cases and include corresponding conditions or verifications on configuration equality. Mutation strategy is of great importance as well. Mutation helps to overcome local optima cases but if the mutation is performed on the entire feasible design set it can break all design solution geometry. Therefore, the developer should process all possible cases of such “wrong” evolution behavior and build corresponding strategies for crossover and mutation operators.

C. Fitness evaluation problem

Evaluation problem addresses the fact that design itself is high-dimensional space and fitness evaluation could depend on design parameters that are not encoded in the chromosome.

In the most general case, the fitness of generated individual determines the probability of its survival for the next generation. This probability depends on the performance and quality of the generated solution, which are defined by user metrics. Evaluation of fitness on particular metric could require additional design parameters not encoded in chromosome or usage of special software. Therefore, it is more flexible to compute fitness function not on the level of genotype (chromosome) but on the level of phenotype (design).

D. Complexity and performance

In fact, the complexity of the generative process itself depends on many parameters including hyper-parameters of the evolution process, the number of evolution epochs, the complexity of evaluation methods, the number of metrics, etc.

While fitness evaluation could require including special third-party software or methods into evolution process, the complexity of the entire generative process could grow dramatically.

Moreover, some of the parameters could stay uncertain during all generative process. For example, the quantity of evolution epochs depends on termination criteria of the evolution process, which could turn to be unreachable. The upper bound for that parameter which is usually defined by maximum iterations limit could archive hundreds of thousand iterations or more.

Therefore, evaluation of generative method performance is possible only when the particular generative case is considered.

IV. METHODOLOGY

Workflow of generative architecture design proposed in this paper is organized into the following steps (see Figure 1).

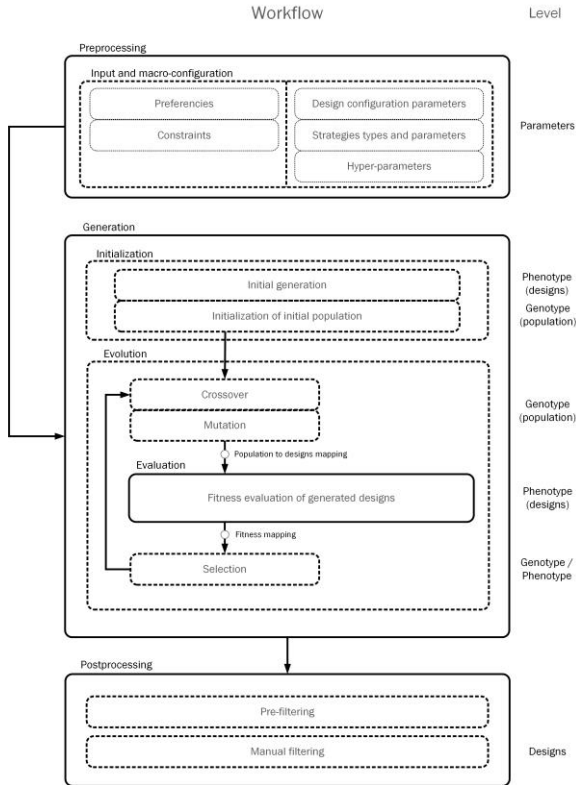


Fig. 1. Proposed generative design workflow

A. Preprocessing

Preprocessing stage is crucial for the success of generative design system development. This step contains input of different user parameters and following generative process configuration.

Input parameters could be divided into the following groups:

1) *Design preferences*: This group of parameters includes preferences resulting from requirements specification and stakeholders expectations. For example, recommended planning, desired number of storeys, etc.

2) *Design metric constraints*: This group of parameters include metric constraints inferring from requirements specification. Research [3] propose the following classification for architectural performance metrics:

- Those that can be easily quantified and calculated using existing tools (e.g. daylight analysis);

- Those that can theoretically be quantified but cannot be computed using existing tools, for which new computation tools must be developed (e.g. work style preference and activity hotspots);
- Those that cannot be quantified and must be addressed through other means outside of generative design (e.g. beauty).

3) *Design configuration parameters*: This group includes constraint parameters that describe desired design solution. For instance, zones types and placement, parametrization preferences and strategies, etc. Geometry configuration parameters could be presented as initial solution model file. In this case, corresponding file parsers should be provided. For instance, IFC [7] parser.

4) *Strategies types and parameters*: This group includes parameters defining types of crossover, mutation, selection and other strategies and their corresponding parameters.

5) *Generation hyper-parameters*: This group includes hyper-parameters of evolution strategy, for instance, population size, termination criteria, etc.

Combination of described input data defines limitations of the entire generative process. It is not mandatory to define all of the listed parameters in a macro-configuration way, but all of them influence the results of generative design and stay accountable.

The way this macro-configuration is provided depends on the purpose of developing a generative system and BIM software this system is integrated with.

B. Initialization

Initialization should be performed both on phenotype and genotype levels. Actually, parametrization strategy defines initialization strategy, and different generative process goals could lead to different parametrization strategies. Various segmentation methods could be used for design initialization: nearest neighbor methods (e.g. Voronov diagram), reference geometry markup, etc.

When design initialization is done, generated phenotypes should be mapped to initial population of chromosomes. Separation of phenotype and genotype initialization is important because design initialization relates to the parametrization of design geometry while genotype initialization usually uses resulting parameters. The parameters define chromosomes and genes structure.

Therefore, building initialization module developer should use flexible mechanisms and approaches to archive a certain level of generalization.

C. Evolution

Combination of crossover, mutation and selection strategies defines evolution strategy. While generative design task could require optimizing design based on number of metrics, different multi-objective evolutionary strategies and algorithms could be used: Multi-objective Genetic Algorithm

(MOGA) [8], Niched Pareto Genetic Algorithm (NPGA) [9], Weight-based Genetic Algorithm (WBGGA) [10], Random Weighted Genetic Algorithm (RWGA) [11], Nondominated Sorting Genetic Algorithm (NSGA) [12], Strength Pareto Evolutionary Algorithm (SPEA) [13], improved SPEA (SPEA2) [14], Pareto-Archived Evolution Strategy (PAES) [15], Pareto Envelope-based Selection Algorithm (PESA) [16], Region-based Selection in Evolutionary Multiobjective Optimization (PESA-II) [17], Fast Nondominated Sorting Genetic Algorithm (NSGA-II) [18], Multi-objective Evolutionary Algorithm (MEA) [19], Micro-GA [20], Rank-Density Based Genetic Algorithm (RDGA) [21], and Dynamic Multi-objective Evolutionary Algorithm (DMOEA) [22]. Research [4] provides a comparison of the most well-known multi-objective genetic algorithms.

Despite the chosen evolution strategy, crossover and mutation operations could lead to different design anomalies. In this paper, the following notion is used.

Two chromosomes are *compatible* with each other if and only if they have equal structure and encode equal configuration.

Term structure relates to the genotype level and means using encoding schema. Term configuration relates to the phenotype level and means configuration of the design solution. For instance, two chromosomes with the same configuration (e.g. number and placement of rooms) could be encoded using different schemas. Similarly, two chromosomes, which encode different configuration, could use the same encoding schema. In both cases – chromosomes are not compatible. Crossover of chromosomes that are not compatible could lead to unpredictable or even broke design geometry.

There are different selection procedures in genetic algorithms depending on how the fitness values are used. Proportional selection, ranking, and tournament selection are the most popular selection procedures. For generative design task additional selection filtering of designs could be used to increase generation stability.

D. Fitness evaluation

Fitness evaluation process depends on metric constraints defined by user input. Depend on metric type evaluation process could require integration with special software and libraries.

Developing evaluation module questions of performance and efficiency are of great importance. Usually, each metric is independent and can be computed using parallel computations.

Usage of special software leads to the problem of mapping mechanisms development for each type of third-party software. These procedures also increase the evaluation step complexity.

Some of design metrics could not be measured by existing engineering tools and could require the development of new one. A good example of such metric is beauty or aesthetics. Application of machine learning methods for this task requires further research.

E. Postprocessing

Postprocessing stage includes automated pre-filtering and manual filtering steps.

Manual filtering is just a process of exploring generated solutions by the designer using different filtering tools. However, resulting solutions could contain many “similar” designs or even duplicates depending on evolution selection strategy. Therefore, an additional step should be used to reduce complexity and quantity of generated solutions. Pre-filtering helps to solve this task.

This step also required integration with parent BIM software to export resulting designs in the appropriate format, for example, IFC [7].

V. CONCLUSION

This paper described architectural generative design process and provided some systematization and generalization on the development of custom generative design method. Main problems behind the implementation of generative evolutionary concepts on different stages were discussed.

Developing universal architectural generative design method is a complicated task, and macro-configuration of the generative process through parameters input is crucial for the success of generative design software development. It is not mandatory to define all of the existing parameters in the macro-configuration way, but all of them influence the results of generative design and stay accountable.

Parametrization strategy and encoding schema are connected. Together they form core element of evolution – genotype. Genotype and phenotype should be considered separately on different evolution stages because it makes the generative process more flexible. Fitness evaluation could be challenging because of development of custom evaluation methods. Moreover, it influences the complexity and performance of entire evolution process to great extent.

Finally, the proposed workflow could be used as a framework for the development of custom generative design methods for new generation of BIM solutions.

REFERENCES

- [1] Abrishami, S., Goulding, J., Rahimian, F. P., & Ganah, A. (2014). Integration of BIM and generative design to exploit AEC conceptual design innovation. *Information Technology in Construction*, 19, 350-359.
- [2] Marler, R. T., & Arora, J. S. (2004). Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6), 369-395.
- [3] Nagy, D., Lau, D., Locke, J., Stoddart, J., Villaggi, L., Wang, R., ... & Benjamin, D. (2017). Project Discover: An Application of Generative Design for Architectural Space Planning. In *Symposium on Simulation for Architecture and Urban Design*.
- [4] Konak, A., Coit, D. W., & Smith, A. E. (2006). Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9), 992-1007.
- [5] Holland JH. *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press; 1975.

- [6] Berquist, J., Tessier, A., O'Brien, W., Attar, R., & Khan, A. An Investigation of Generative Design for Heating, Ventilation, and Air-Conditioning.
- [7] IFC4 Documentation. URL: <http://www.buildingsmart-tech.org/ifc/IFC4/final/html/>
- [8] Fonseca CM, Fleming PJ. Multiobjective genetic algorithms. In: IEE colloquium on 'Genetic Algorithms for Control Systems Engineering' (Digest No. 1993/130), 28 May 1993. London, UK: IEE; 1993.
- [9] Horn J, Nafpliotis N, Goldberg DE. A niched Pareto genetic algorithm for multiobjective optimization. In: Proceedings of the first IEEE conference on evolutionary computation. IEEE world congress on computational intelligence, 27–29 June, 1994. Orlando, FL, USA: IEEE; 1994.
- [10] Hajela P, Lin C-y. Genetic search strategies in multicriterion optimal design. *Struct Optimization* 1992;4(2):99–107.
- [11] Murata T, Ishibuchi H. MOGA: multi-objective genetic algorithms. In: Proceedings of the 1995 IEEE international conference on evolutionary computation, 29 November–1 December, 1995. Perth, WA, Australia: IEEE; 1995.
- [12] Srinivas N, Deb K. Multiobjective optimization using nondominated sorting in genetic algorithms. *J Evol Comput* 1994;2(3):221–48.
- [13] Zitzler E, Thiele L. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans Evol Comput* 1999;3(4):257–71.
- [14] Zitzler E, Laumanns M, Thiele L. SPEA2: improving the strength Pareto evolutionary algorithm. Swiss Federal Institute Technology: Zurich, Switzerland; 2001.
- [15] Knowles JD, Corne DW. Approximating the nondominated front using the Pareto archived evolution strategy. *Evol Comput* 2000;8(2):149–72.
- [16] Corne DW, Knowles JD, Oates MJ. The Pareto envelope-based selection algorithm for multiobjective optimization. In: Proceedings of sixth international conference on parallel problem solving from Nature, 18–20 September, 2000. Paris, France: Springer; 2000.
- [17] Corne D, Jerram NR, Knowles J, Oates J. PESA-II: region-based selection in evolutionary multiobjective optimization. In: Proceedings of the genetic and evolutionary computation conference (GECCO-2001), San Francisco, CA, 2001.
- [18] Deb K, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput* 2002;6(2):182–97.
- [19] Sarker R, Liang K-H, Newton C. A new multiobjective evolutionary algorithm. *Eur J Oper Res* 2002;140(1):12–23.
- [20] Coello CAC, Pulido GT. A micro-genetic algorithm for multiobjective optimization. In: Evolutionary multi-criterion optimization. First international conference, EMO 2001, 7–9 March, 2001. Zurich, Switzerland: Springer; 2001.
- [21] Lu H, Yen GG. Rank-density-based multiobjective genetic algorithm and benchmark test function study. *IEEE Trans Evol Comput* 2003;7(4):325–43.
- [22] Yen GG, Lu H. Dynamic multiobjective evolutionary algorithm: adaptive cell-based rank and density estimation. *IEEE Trans Evol Comput* 2003;7(3):253–74.

Product Reviews Sentiment Analysis in Russian

Sergey Smetanin
Faculty of Business and Management
National Research University
Higher School of Economics
Moscow, Russia
sismetanin@gmail.com

Mikhail Komarov
Faculty of Business and Management
National Research University
Higher School of Economics
Moscow, Russia
mkomarov@hse.ru

Abstract — Nowadays reviews on e-commerce sites that generated by customers tend to be a valuable resource in terms of evaluation of customers' behavior, their preferences and needs. This paper provides an approach for sentiment analysis of product reviews in Russian using a convolutional neural network. The training dataset was collected from reviews on Aliexpress top-ranked goods, where the user-ranked score was used as a class label on a 5-point scale. Word2Vec was used in order to set up pre-trained word embeddings for one-layer convolutional neural network, which was constructed using Keras with Theano backend. Experiments showed F-measure score up to 52.51%.

Keywords—*sentimen analysis; convolutional neural network; natural language processing.*

I. INTRODUCTION

It is a common practice for users of the e-commerce sites not only to analyze recommendations of users, which already have bought a specific product, but also to post reviews of the products that they purchase. As online shopping is becoming more and more popular, a volume of user-generated textual content increase dramatically.

This paper major goal is to present the proposed Convolutional Neural Network (CNN) architecture for sentiment analysis of product reviews in Russian based on pre-trained word embeddings, obtained using Word2Vec. The training data for the classifier were collected from the e-commerce site Aliexpress [4].

The rest of the article is organized as follows. Section 2 lays emphasis on the overview of the related work. Section 3 is focused on the data collection process. Pre-processing and word embedding techniques are provided in section 4. In section 5 multinomial Naïve Bayes classifier as a baseline model is described. The CNN architecture is proposed in section 6. In the conclusion, the results

and further ways of research of this paper are discussed.

II. RELATED WORK

Machine learning techniques for sentiment analysis of product reviews tend to be a common topic in natural language processing sphere. Basic classification algorithms for sentiment analysis such as Naïve Bayes and decision list classifier were described in many top-cited papers [23] [12] [20], so it was decided to use it as a baseline solution.

In recent years deep learning techniques have captured the attention of researchers due to their ability to outperform significantly traditional methods. To explore state-of-the-art solutions, the top-ranked evaluations of computational semantic analysis systems for English and Russian were examined in the first order.

The first competition for predominantly English language is SemEval-2017, which was organized under the umbrella of SIGLEX, the Special Interest Group on the Lexicon of the Association for Computational Linguistics. The paper [10] describes the approach, which won all five subtasks in SemEval-2017 Task 4 [29], which are focused in sentiment analysis. The author pre-trained word embeddings on a large amount of unlabeled data using Word2Vec [17], FastText [7], Glove [22], and then fine-tuned obtained representations using distant supervision. CNNs and Long Short Term Memory (LSTMs) were finally trained on the dataset provided by SemEval-2017. In addition to separate performance evaluation, several CNNs and LSTMs were combined to boost classification performance.

The second one is SentiRuEval-2016 Sentiment Analysis Task, which was organized as a special

direction of the International Conference on Computational Linguistics and Intellectual Technologies «Dialogue-2016». LSTMs and CNNs applications to the aspect-based sentiment analysis task were explored in the paper [27], which were selected as the best model at SentiRuEval-2016 [16]. The authors described approaches based on Gated Recurrent Unit neural network (GRU), CNN, and SVM, which also used Word2Vec vectors as word embeddings. The GRU-based approach shows the best results in terms of macro-averaged F1-score.

Deep CNNs, which were proposed in [11], jointly uses character-level, word-level and sentence-level features to perform sentiment analysis of short texts. This approach achieved 85.7% and 86.4% accuracy in binary classification task for SSTb corpus [26] and STS corpus [12], respectively, and 48.3% accuracy in 5-class classification for SSTb corpus.

The effect of architecture components on CNN model performance was conducted in [29]. Authors proposed a one-layer CNN architecture for sentence classification, where simultaneously used multiple filters for the same region size and multiple kinds of filters with different region sizes. As a result, authors summarized their experience into a practitioners' guide to CNN for sentiment classification at a sentence level.

According to the [15], the convolutional neural networks should not be complex to realize strong results. A simple one-layer CNN demonstrated state-of-the-art accuracy results for English language datasets: 81.5% for binary movie reviews dataset with one sentence per review [21], 88.1% for SST-2 corpus, 85.0% for binary customer reviews dataset [13], and 89.6% for opinion polarity detection subtask of the MPQA dataset [28].

Based on the analyzed papers it was decided to use CNN for Russian language because of the ability to provide state-of-the art results without complex structure. The proposed CNN architecture was inspired by approaches described in [10] and [29]. The Word2Vec representation for initial word embeddings was chosen based on information provided by papers [25] and [16].

III. DATA COLLECTION

The training dataset was collected from reviews on Aliexpress top-ranked goods, where the user-ranked score was used as a class label on a 5-point scale. The

Aliexpress Reviews Dataset was collected from the all top-level product categories from the official website. It has 4529391 samples.

According to the obtained data, in some cases, it is not only difficult to distinguish reviews score with the close values (e.g. with score 1 and 2 or with score 4 and 5) but also to correctly evaluate reviews with the same texts. For example, the review text "Нормально" ("Fine" translation from Russian into English) has 42.45% reviews with a score of 5, 27.36% reviews with a score of 4, 29.25% reviews with a score of 3, and 0.94% reviews with a score of 2. It's clear that such contradictions in the training dataset tend to affect the classification score, so we decided to mark these reviews with the class label, which is the most common for this text in the collected dataset. Coming back to the example mentioned above, all reviews with such text should be labelled with a score of 5 because it occurs in 42.45% of cases.

After such type of preprocessing, we found, that data are quite imbalanced, so we decided to use an undersampling technique to solve this problem. For each class 30000 of reviews were randomly selected to make balanced dataset.

IV. WORD EMBEDDINGS

A. Text Pre-processing

Before any training stage, reviews were pre-processed using the following procedure:

- All characters in the text are converted to lowercase.
- All non-alphabet characters, except numbers, emoji, and some punctuation marks ("+", "-", "%"), are replaced with space.
- Emoji are replaced by the tokens defined by the Unicode consortium [2], e.g. "simple_smile", "laughing", "thumbsdown", "disappointed_relieved" and so on. Emoji for Python library [1] was used to extract emoji from raw texts and convert them to their textual annotation.
- Repeated letters, which occurs together more than 2 times in a row, were replaced by 2 such letters. For instance, "so coool" was replaced with "so cool".

B. Word2Vec

Word2vec [17] [18] [19] is a computationally-efficient predictive model for learning low-dimensional word embeddings from raw textual data. It consists of the Continuous Bag-of-Words model (CBOW) and the Skip-Gram model. They are quite similar, except the CBOW predicts the target word according to the current context, and the Skip-Gram does the inverse task, i.e. predicts the context according to the current target word. Word vectors that were calculated using Word2Vec have been shown to capture semantic information. Thus, their usage in many NLP tasks leads to major improvements.

We used Gensim [24] with Word2Vec support for obtaining vector representations of Russian words. The model was trained on the entire dataset, which was collected and pre-processed at the previous steps. Each review was tokenized into sentences using NLTK [3]. The following training parameters were used based on [27]:

- CBOW architecture with negative sampling;
- dimensionality of the feature vector is 200;
- the maximum distance between analyzed words within a sentence is 5;
- number of epochs over the corpus is 5;
- ignores all words with the total frequency lower than 2 per corpus.

V. BASELINE MODEL

The multinomial Naïve Bayes Classifier was selected as a baseline classification algorithm because of its tendency to perform significantly well in the sentiment analysis task [14]. The basic idea of Naïve Bayes technique is to find the probabilities of classes assigned to texts by using the joint probabilities of words and classes. Consider the given data point x and class $c \in C$. The starting point is Bayes' theorem for conditional probability which estimates as follows:

$$P(c|x) = \frac{P(x|c)}{P(x)} \quad (1)$$

$$P(x|c) = \frac{\text{count}(x,c)}{\text{count}(c)} \quad (2)$$

Where $\text{count}(x, c)$ is the count of word x in class c ; $\text{count}(c)$ is a count of all words in class c . For texts with unknown words, the estimation (2) might be problematic because it would give zero probability. The usage of Laplace smoothing is a common way to solve this problem (3).

$$P(x|c) = \frac{\text{count}(x,c)+1}{\text{count}(c)+|V|+1} \quad (3)$$

Where $|V|$ is the length of vocabulary in training set.

From the assumption of word independence, it appears that for data point $x = \{x_1, x_2, \dots, x_i\}$ the probability of each of its features to occur in the given class is independent. Thus, the estimation of this probability can be calculated as follows:

$$P(c|x) = P(c) \prod P(x_i|c) \quad (4)$$

In this context, that means the final equation for the class chosen by a naive Bayes classifier is (5).

$$c_{nb} = \underset{c \in C}{\operatorname{argmax}} P(c) \prod P(x_i|c) \quad (5)$$

To avoid underflow and increase speed, the Naive Bayes calculations are performed in the log space (6).

$$c_{nb} = \underset{c \in C}{\operatorname{argmax}} (\log P(c) + \sum \log P(x_i|c)) \quad (6)$$

The classifier was implemented using machine learning library Scikit-Learn [5]. The Naïve Bayes classifier was trained on the balanced dataset with 150000 reviews. The grid search was used to optimize following training parameters [9]: n-gram range, use of TF-IDF, TF-IDF normalization, use of Additive (Laplace/Lidstone) smoothing. The dataset was split into random train (70% of the entire dataset) and test subset (30% of the entire dataset). The 10-fold cross-validation shows accuracy on the test subset up to 47.12%.

VI. CONVOLUTIONAL NEURAL NETWORK

A. CNN Architecture

The proposed CNN architecture was inspired by approaches described in [10] and [29]. To begin with, a tokenized sentence converted to a sentence matrix, where each row represents a word vector. In our case, this representation is outputs from the trained Word2Vec model, where the dimensionality of words

vector is $d = 200$. Assuming that a length of a given sentence is s (based on our dataset we decided to use $s = 60$), the dimensionality of a sentence matrix is $s \times d$.

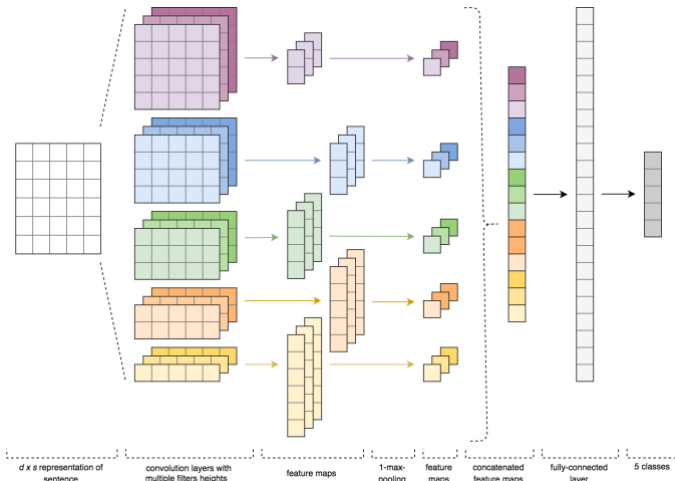


Fig. 1. The simplified illustration of a CNN architecture for review classification. There are five filter region sizes: 1, 2, 3, 4 and 5, each of which has 3 filters. These filters generate feature maps by performing convolutions on the sentence matrix. Next 1-max pooling performed on each feature map and results are concatenated into one vector. Concatenated features vector goes through the fully-connected hidden layer and at the final stage it processed by the softmax activation layer.

A configuration of different filter sizes and their amount affect significantly the classification quality. For example, multiple filters with the same region sizes provide with complementary features of the analyzed data, whereas multiple types of filters with a variety of region sizes allow focusing on smaller or larger regions of the texts. We use five filter sizes [1, 2, 3, 4, 5] with a total of 50, 50, 50, 30, 20 filtering matrices (ReLU activation function) for each convolution filter size (based on [10] and [29]), consequently. It should be mentioned that significant part of the reviews dataset contains texts with only one token (an emoticon or a word). Thus, it was decided to use filters size of 1 in order to correctly extract information from these reviews.

After that, a max-pooling operation is applied to each convolutional layer output in order to extract major features, independently of their location in the text. It can be interpreted as an extraction of the most important n-grams from the text based on their embeddings. Next, maximum values obtained from each convolutional layer combined into one vector and then processed by fully-connected layers of size 100. At the last step, feature maps from the fully-

connected layer go through a sigmoid activation layer to predict the final classification labels.

To reduce overfitting, dropout layers were added after the max-pooling layer (with dropout probability $p = 0.3$) and after the fully connected layer (with dropout probability $p = 0.1$).

B. Supervised Training

At the training stage, the balanced dataset with 150000 reviews was used. It was divided into three parts: train dataset (60% of the entire dataset), validation dataset (20% of the entire dataset), and test dataset (20% of the entire dataset). The loss function was minimized using the Adam optimizer with a learning rate of 0.001. The embedding layer was initialized with Word2Vec word embeddings and was frozen for the first 10 epochs. Then we train model from the previous step with best validation scores (that was the model obtained at the 5 epoch) for additional 10 epochs with unfrozen embeddings. The best results on the validation subset was obtained at the 3 epochs and in terms of F-measure was 52.80%. This model was evaluated on the test dataset and demonstrated F-measure score up to 52.51%.

The models were implemented using Keras [8] with Theano [6] backend. Experiments were run on AWS EC2 GPU optimized instance g2.2xlarge (1 NVIDIA GRID K520 Kepler GPU).

VII. RESULTS

We have proposed CNN architecture for reviews classification in Russia and have trained it on the collected dataset. Preprocessed reviews were mapped into feature space using Wor2Vec. The best validation score for CNN in terms of F-measure was 52.51%, while MNB demonstrated F-measure value up to 47.12%. Thus, CNN significantly outperformed the baseline approach.

The data, which was obtained in the research, is available by the URL <https://bitbucket.org/sismetanin/aliexpress-reviews-dataset>.

The further research will be focused on adding a second layer of hidden filters in the CNN architecture in order to extract and process more complex syntactic and semantic features.

REFERENCES

- [1] "Emoji for Python", Python, 2018. [Online]. Available: <https://pypi.python.org/pypi/emoji/>. [Accessed: 25- Mar- 2018].
- [2] "Full Emoji List, v11.0", The Unicode Consortium, 2018. [Online]. Available: <http://www.unicode.org/emoji/charts/full-emoji-list.html>. [Accessed: 25- Mar- 2018].
- [3] "NLTK documentation", Natural Language Toolkit, 2018. [Online]. Available: <https://www.nltk.org/>. [Accessed: 25- Mar- 2018].
- [4] "Products from China Wholesalers at Aliexpress", Aliexpress, 2018. [Online]. Available: <https://www.aliexpress.com>. [Accessed: 25- Mar- 2018].
- [5] "Scikit-Learn: Machine Learning in Python", Scikit-Learn, 2018. [Online]. Available: <http://scikit-learn.org/stable/>. [Accessed: 25- Mar- 2018].
- [6] Bergstra J. et al. "Theano: A CPU and GPU math compiler in Python", *Proceedings of the Python for Scientific Computing Conference (SciPy)*, pp. 1-7, 2010.
- [7] Bojanowski P., Grave E., Joulin A., Mikolov T. "Enriching word vectors with subword information," *arXiv preprint arXiv:1607.04606*, 2016.
- [8] Chollet F. *Keras*. 2015.
- [9] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, *Introduction to Information Retrieval*. Cambridge University Press. 2008.
- [10] Cliche M. "BB_twtr at SemEval-2017 Task 4: Twitter Sentiment Analysis with CNNs and LSTMs", *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pp. 572–579, 2017.
- [11] dos Santos C., Gatti M. "Deep convolutional neural networks for sentiment analysis of short texts", *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pp. 69–78, 2014.
- [12] Go A., Bhayani R., Huang L. "Twitter Sentiment Classification using Distant Supervision," *CS224N Project Report*, pp. 1-12, 2009.
- [13] Hu M., Liu B. "Mining and summarizing customer reviews," *Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 168-177, 2004.
- [14] Jurafsky D. Martin J. "Naive Bayes and Sentiment Classification," in *Speech and Language Processing*, draft of August 7, 2017.
- [15] Kim Y. "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.
- [16] Loukachevitch N. V., Rubtsova Y. V. "SentiRuEval-2016: overcoming time gap and data sparsity in tweet sentiment analysis," *Computational Linguistics and Intellectual Technologies: Proceedings of the International Conference Dialogue*, pp. 375–384, 2016.
- [17] Mikolov T., Chen K., Corrado G., Dean J. "Efficient estimation of word representations in vector space", *arXiv preprint arXiv:1301.3781*, 2013.
- [18] Mikolov T., Sutskever I., Chen K., Corrado G. S., Dean J. "Distributed representations of words and phrases and their compositionality", *Advances in Neural Information Processing Systems*, pp. 3111-3119, 2013.
- [19] Mikolov T., Yih W., Zweig G. "Linguistic regularities in continuous space word representations", *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 746-751, 2013.
- [20] Pak A., Paroubek P. "Twitter as a Corpus for Sentiment Analysis and Opinion Mining," *Proceedings of the Seventh Conference on International Language Resources and Evaluation*, pp. 1320–1326, 2010.
- [21] Pang B., Lee L. "Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales," *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2005.
- [22] Pennington J., Socher R., Manning C. "Glove: Global vectors for word representation," *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532-1543, 2014.
- [23] Rain C. "Sentiment Analysis in Amazon Reviews Using Probabilistic Machine Learning", *Swarthmore College*, 2013.
- [24] Rehurek R., Sojka P. "Software framework for topic modelling with large corpora", *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pp. 46-50, 2010.
- [25] Rosenthal S., Farra N., Nakov P. "SemEval-2017 task 4: Sentiment analysis in Twitter" *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pp. 502–518, 2017.
- [26] Socher R., Perelygin A., Wu J., Chuang J., Manning C. D., Ng A., Potts C. "Recursive deep models for semantic compositionality over a sentiment treebank," *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1631-1642, 2013.
- [27] Trofimovich J., "Comparison of Neural Network Architectures for Sentiment Analysis of Russian Tweets", *Computational Linguistics and Intellectual Technologies: Proceedings of the International Conference Dialogue*, pp. 50–59, 2016.
- [28] Wiebe J., Wilson T., Cardie C., "Annotating Expressions of Opinions and Emotions in Language", *Language Resources and Evaluation*, vol. 39, no. 2-3, pp. 165-210, 2005.
- [29] Zhang Y., Wallace B. "A sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification," *arXiv preprint arXiv:1510.03820*, 2015.

On the verification of strictly deterministic behaviour of Timed Finite State Machines

Evgenii Vinarskii

Lomonosov Moscow State University

Moscow, Russia

vinevg2015@gmail.com

Vladimir Zakharov

Lomonosov Moscow State University

Moscow, Russia

zakh@cs.msu.su

Abstract—Finite State Machines (FSMs) are widely used as formal models for solving numerous tasks in software engineering, VLSI design, development of telecommunication systems, etc. To describe the behavior of a real-time system one could supply FSM model with clocks — a continuous time parameters with real values. In a Timed FSM (TFSM) inputs and outputs have timestamps, and each transition is equipped with a timed guard and an output delay to indicate time interval when the transition is active and how much time does it take to produce an output. A variety of algorithms for equivalence checking, minimization and test generation were developed for TFSMs in many papers. A distinguishing feature of TFSMs studied in these papers is that the order in which output letters occur in an output timed word does not depend on their timestamps. We think that such behaviour of a TFSM is not realistic from the point of view of an outside observer. In this paper we consider a more advanced and adequate TFSM functioning; in our model the order in which outputs become visible to an outsider is determined not only by the order of inputs, but also by delays required for their processing. When the same sequence of transitions is performed by a TFSM modified in a such way, the same outputs may follow in different order depending on the time when corresponding inputs become available to the machine. A TFSM is called strictly deterministic if every input timed word activates no more than one sequence of transitions (trace) and for any input timed word which activates this trace the letters in the output words always follows in the same order (but, maybe, with different timestamps). We studied the problem of checking whether a behaviour of an improved model of TFSM is strictly deterministic. To this end we showed how to verify whether an arbitrary given trace in a TFSM is steady, i.e. preserves the same order of output letters for every input timed word which activates this trace. Further, having the criterion of trace steadiness, we developed an exhaustive algorithm for checking the property of strict determinacy of TFSMs. Exhaustive search in this case can hardly be avoided: we proved that determinacy checking problem for our model of TFSM is co-NP-hard.

I. INTRODUCTION

Finite State Machines (FSMs) are widely used as formal models for analysis and synthesis of information processing systems in software engineering, VLSI design, telecommunication, etc. The most attractive feature of this model of computation is its simplicity — many important synthesis and analysis problems (equivalence checking, minimization, test derivation, etc.) for classical FSMs can be solved in time which is almost linear or quadratic of the size of a FSM under consideration.

The concept of FSM is rather flexible. Since in many applications time aspects such as durations, delays, timeouts are very important, FSMs can be augmented with some additional features to describe the dependence of the behavior of a system on events occurring in real time. One of the most advanced timed extension of FSMs is the concept of Timed Automata which was developed and studied in [1]. Timed Automata are supplied with clocks (timers) for indicating real time moments, measuring durations of events, providing timeout effects. Transitions in such automata depends not only on the incoming of the outside messages and signals but also on the values of clocks. Further research showed that this model of computation is very expressive and captures many important features of real-time systems behaviour. On the other side, Timed Automata in the full scope of their computing power are very hard for analysis and transformations. The reachability problem for Timed Automata is decidable [2], and, therefore, this model of computation is suitable for formal verification of real-time computer systems. But many other problems such as universality, inclusion, determinizability, etc. are undecidable (see [2], [8]), and this hampers considerably formal analysis of Timed Automata.

When a Timed Automaton is capable to selectively reset timers it can display rather sophisticated behaviour which is very difficult for understanding and analysis. In some cases such ability is very important; see, e.g. [9]. But a great deal of real-time programs and devices operate with timers much more simply: as soon as such a device switches to a new mode of operation (new state), it resets all timers. Timed Finite State Machines (TFSM) of this kind were studied in [5], [10], [13], [14]. TFSM has the only timer which it resets "automatically" as soon as it moves from one state to another. On the other hand, TFSMs, in contrast to Timed Automata introduced in [1], operate like transducers: they receive a sequence of input signals augmented with their timestamps (input timed word) and output a sequence of responses also labeled by timestamps (output timed word). The timestamps are real numbers which indicate the time when an input signal becomes available to a TFSM or an output response is generated. Transitions of a TFSM are equipped with time guards to indicate time intervals when transitions are active. Therefore, a reaction of a TFSM to an input signal depends not only on the signal but also on its timestamp. Some algorithms for equivalence checking,

minimization and test generation were developed for TFSMs in [6], [5], [13], [14], [15]. It can be recognized that this model of TFSM combines a sufficient expressive power for modeling a wide class of real-time information processing systems and a developed algorithmic support.

As it was noticed above a behaviour of a TFSM is characterized by a pair sequences: an input timed word and a corresponding output timed word. A distinguishing feature of TFSMs studied in [5], [10], [13], [14], [15] is that an output timed word is formed of timestamped output letters that follows in the same order as the corresponding input letters regardless of their timestamps. Meanwhile, suppose that a user of some file management system gives a command "Save" and immediately after that a command "Exit". Then if a file to be saved is small then the user will observe first a response "File is saved" and then a notification "File Management System is closed". But if a file has a considerable size then it takes a lot of time to close it. Therefore, it can happen that a user will detect first a notification "File Management System is closed" and then, some time later, he/she will be surprised to find an announcement "File is saved". Of course, the user may regard such behaviour of the system enigmatic. But much worse if the order in which these notifications appear may vary in different sessions of the system. If a File Management System interacts with other service programs such an interaction will almost certainly lead to errors. However, if a behaviour of TFSMs is defined as in the papers referred above then such a model can not adequately capture behavioral defects of real-time systems, similar to the one that was considered in the example.

To avoid this shortcoming of conventional TFSMs and to make their behaviour more "realistic" from the point of view of an outside observer we offer some technical change to this model. We will assume that an output timed word consists of timestamped letters, and these letters always follow in ascending order of their timestamps regardless of an order in which the corresponding input letters entered a TFSM. In this model it may happen so that an input b follows an input a but a response to b appears before a response to a is computed. Clearly, the defect with File Management System discussed above becomes visible to an outside observer "through" the model of TFSMs thus modified.

At first sight, it may seem that this change only slightly complicates the analysis of the behavior of such models. But this is a false impression. In the initial model of TFSM the formation of an output timed word is carried out by local means for each state of the system. In our model this is a global task since to find the proper position of a timestamped output letter one should consider the run of TFSM as a whole. Therefore, even the problem of checking whether a behaviour of an improved model of TFSM is deterministic can not be solved as easy and straightforwardly as in the case of the initial model of TFSM.

It should be noticed that the property of deterministic behavior is very important in theory real-time machines. As it was said above, universality, inclusion and equivalence checking problems are undecidable for Timed Automata in

general case [2] but all these problems have been shown to be decidable for deterministic Timed Automata [3], [11]. However, testing whether a Timed Automaton is determinizable has been proved undecidable [8]. Understanding and coping with these weaknesses have attracted lots of research, and classes of timed automata have been exhibited, that can be effectively determinized [3], [12]. A generic construction that is applicable to every Timed Automaton, and which, under certain conditions, yields a deterministic Timed Automaton, which is language-equivalent to the original timed automaton, has been developed in [4].

We studied the determinacy checking problem for improved TFSMs and present the results of our research in this paper. First, we offer a criterion to determine whether a given sequence of transition (trace) in a TFSM is steady, i.e. for any input timed word which activates this trace the letters of output words always follow in the same order (but, maybe, with different timestamps). Then, using this criterion we developed an exhaustive algorithm for checking the property of strict determinacy of TFSMs. This property means that every input timed word activates no more than one trace and all traces in a TFSM are steady. Exhaustive search, although been time consuming, can hardly be avoided in this case: we proved that determinacy checking problem for improved version of TFSMs is co-NP-hard by polynomially reducing to its complement the subset-sum problem [7] which is known to be NP-complete.

The structure of the paper is as follows. In Section II we define the basic notions and introduce an improved concept of TFSM (or, it would be better said, a concept of TFSM with an improved behaviour). In Section III we present necessary and sufficient conditions for steadiness of traces in a TFSM and show how to use this criterion to check whether a given TFSM is strictly deterministic. Section IV contains the results on the complexity of checking the properties of strictly deterministic behavior of TFSM. In the Conclusion we briefly outline the consequences of our results and topics for further research.

II. PRELIMINARIES

Consider two non-empty finite alphabets I and O ; the alphabet I is an *input alphabet* and the alphabet O is an *output alphabet*. The letters from I can be regarded as control signals received by some real-time computing system, whereas the letters from O may be viewed as responses (actions) generated by the system. A finite sequence $w = i_1, i_2, \dots, i_n$ of input letters is called an *input word*, whereas a sequence $z = o_1, o_2, \dots, o_n$ of output letters is called an *output word*. As usual, the time domain is represented by the set of non-negative reals \mathbb{R}_0^+ . The set of all positive real numbers will be denoted by \mathbb{R}^+ . When such a system receives a control signal (a letter i) its output depends not only on the input signal i but also on

- a current internal state of the system,
- a time instance when i becomes available to a system, and
- time required to process the input (output delay).

These aspects of real-time behaviour can be formalized with the help of timestamps, time guards and delays. A timestamp as well as a delay is a real number from \mathbb{R}^+ . A *timestamp* indicates a time instance when the system receives an input signal or generates a response to it. A *delay* is time the system needs to generate an output response after receiving an input signal. A *time guard* is an interval $g = \langle u, v \rangle$, where $\langle \cdot, \cdot \rangle \in \{(\cdot, \cdot], [\cdot, \cdot), \cdot, \cdot\}$, and u, v are timestamps such that $0 < u < v$. Time intervals indicate the periods of time when transitions of a system are active for processing input signals. As usual, the term *time sequences* is reserved for an increasing sequence of timestamps. For the sake of simplicity we will deal only with time guards of the form $(u, v]$: all the results obtained in this paper can be adapted with minor changes to arbitrary time guards.

Let $w = x_1, x_2, \dots, x_n$ and $\tau = t_1, t_2, \dots, t_n$ be an input (output) word and a time sequence, respectively, of the same length. Then a pair (w, τ) is called a *timed word*. Every pair of corresponding elements x_j and t_j , $1 \leq j \leq n$, indicates that an input signal (or an output response) x_j appears at time instance t_j . In order to make this correspondence more clear we will often write timed words as sequences of pairs $(w, \tau) = (i_1, t_1), (i_2, t_2), \dots, (i_n, t_n)$ whose components are input signals (or output responses) and their timestamps.

A *Finite State Machine (FSM)* over the alphabets I and O is a triple $M = \langle S, s_{in}, \rho \rangle$ where

- S is a finite non-empty set of *states*,
- s_{in} is an *initial state*,
- $\rho \subseteq (S \times I \times O \times S)$ is a *transition relation*.

A transition (s, i, o, s') means that FSM M when being at the state s and receiving an input signal i moves to the state s' and generates the output response o .

FSMs can not measure time and, therefore, they are unsuitable for modeling the behavior of real-time systems. The authors of [1] proposed to equip FSMs with clocks — variables which take non-negative real values. To manipulate with clocks machines use reset instructions, timed guards and output delays. Time guards indicate time intervals during which transitions are active and input signals can be processed. An output delay indicates how much time does it take to process an input. Thus, every transition in such a machine is a quadruple $\langle \text{input}, \text{timed guard}, \text{output}, \text{delay} \rangle$. Input signals and output responses are accompanied by timestamps. If an *input* is marked by a timestamp which satisfies the *time guard* then the transition fires, the machine moves to the next state and generates the *output*. This output is marked by a timestamp which is equal to the timestamp of the input plus the *delay*. For real-time machines of this kind usual problems from automata theory (equivalence and containment checking, minimization, etc.) may be set up and solved. The minimization problem for real-time machines is very important, since the complexity of many analysis and synthesis algorithms depend on the size of machines. In [14] this problem was studied under the so called "slow environment assumption": next input becomes available only after an output response to the previous one is generated.

In this paper, we consider a more advanced real-time machine; in this model the order in which outputs become visible to an outside observer is determined not only by the order in which inputs follow, but also by the delay required for their processing. When the same sequence of transitions is performed by such a machine the same outputs may follow in different order depending on the arriving time of the corresponding inputs. Our main goal is to develop equivalence checking and minimization algorithms for real-time machines of this kind. But, as the results of Automata Theory show, these problem may have efficient solution only for deterministic machines. Thus, our first step toward the solution of these problems is to find a way to check if the behaviour of a machine is deterministic.

But there is also another reason to study the problem of checking the determinism of the behavior of real-time machines. Unlike traditional discrete models of computation, the behavior of real-time machines depends not only on the control signals as such, but also on the time of their arrival. However, the latter factor has a greater degree of uncertainty. In most cases, in practice, it is desirable to reduce the effect of this uncertainty to a minimum. Therefore, the determinacy checking problem for real-time machines can be considered as a special version of the verification problem — checking that the time factor does not have an unforeseen influence on the behavior of the system.

Formally, by Timed FSM (TFSM) over the alphabets I and O we mean a quadruple $M = (S, s_{in}, G, \rho)$ where:

- S is a finite non-empty set of *states*,
- s_{in} is an *initial state*,
- G is a set of *timed guards*,
- $\rho \subseteq (S \times I \times O \times S \times G \times \mathbb{R}^+)$ is a *transition relation*.

A transition (s, i, o, s', g, d) should be understood as follows. Suppose that TFSM receives the input letter i marked by a timestamp t when being at the state s . If the previous letter has been delivered to the TFSM at time \hat{t} such that $\Delta t = t - \hat{t} \in g$ then the TFSM moves to the state s' and outputs the letter o marked with the timestamp $\tau = t + d$. When algorithmic and complexity issues of TFSM's analysis and synthesis are concerned then we assume that time guards and delays are rational numbers, and the size of a TFSM is the length of a binary string which encodes all transitions in the TFSM.

A *trace* tr in TFSM M is a sequence of transitions $(s_0, a_1, b_1, s_1, (u_1, v_1], d_1), \dots, (s_{n-1}, a_n, b_n, s_n, (u_n, v_n], d_n)$, where every state s_j , $0 < j < n$, is an arrival state of one transition and a departure state of the next transition. We say that the trace tr *converts* an input timed word $\alpha = (a_1, t_1), (a_2, t_2), \dots, (a_n, t_n)$ to the timed output word $\beta = (b_{j_1}, \tau_1), (b_{j_2}, \tau_2), \dots, (b_{j_n}, \tau_n)$, iff

- $t_j - t_{j-1} \in (u_j, v_j]$ holds for all j , $1 \leq i \leq n$ (it is assumed that $t_0 = 0$);
- β is such a permutation of the sequence $\gamma = (b_1, t_1 + d_1), (b_2, t_2 + d_2), \dots, (b_n, t_n + d_n)$ that the second components of the pairs $\tau_1, \tau_2, \dots, \tau_n$ constitute a time sequence.

Clearly, for every trace tr and an input timed word α its conversion β (if any) is determined uniquely; such a conversion will be denoted as $conv(tr, \alpha)$. If $conv(tr, \alpha)$ is defined then we say that the input timed word α *activates* the trace tr . We will say that the output word $b_{j_1}, b_{j_2}, \dots, b_{j_n}$ is a *plain response* to the input timed word α on the trace tr ; it will be denoted as $resp(tr, \alpha)$.

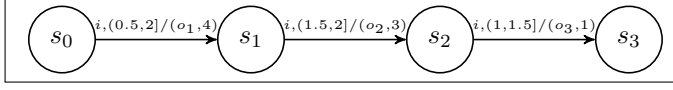


Fig.1 TFSM M

Consider, for example, a TFSM M depicted in Fig. 1 and a trace

$$tr = (s_0, i, s_1, o_1, (0.5, 2], 4), (s_1, i, s_2, o_2, (1.5, 2], 3), (s_2, i, s_3, o_3, (1, 1.5], 1)$$

in this TFSM. Then this trace

- 1) accepts an input timed word $\alpha_1 = (i, 1), (i, 2.7), (i, 4.1)$ and converts it to the output timed word $\beta_1 = (o_1, 5), (o_3, 5.1), (o_2, 5.7)$; thus, the plain response of M to α_1 is $w_1 = o_1, o_3, o_2$;
- 2) accepts an input timed word $\alpha_2 = (i, 1.5), (i, 3.2), (i, 4.3)$ and converts it to the output timed word $\beta_2 = (o_3, 5.3), (o_1, 5.5), (o_2, 6.2)$, and the plain response of M to α_2 is $w_2 = o_3, o_1, o_2$ which is different from w_1 ;
- 3) does not accept an input timed word $\alpha_3 = (i, 2.3), (i, 4), (i, 6)$.

III. STEADY TRACES AND STRICTLY DETERMINISTIC TFSMS

As can be seen from the above example, a pair of input timed words that differ only in timestamps of input signals may activate the same trace in a TFSM, although plain responses of TFSM to these words are different. Generally speaking, there is nothing unusual in this: in real-time models not only the input signals, but also the values of timers influence a run of a model. Nevertheless, in many applications it is critically important to be sure that the behaviour of a real-time system is predictable: once a system choose a mode of computation (i.e. a trace in TFSM) it will behave in a similar way (i.e. give the same plain response) in all computations of this mode. Traditionally, computer systems in which for any input data the processing mode is uniquely determined by the system are called deterministic. But for our model of real-time systems this requirement should be clarified and strengthened. For this purpose we introduce the notion of steady traces and the property of strict determinacy of a real-time system.

A trace tr in TFSM M is called *steady* if $resp(tr, \alpha_1) = resp(tr, \alpha_2)$ holds for every pair of input timed words α_1 and α_2 that activate tr . Thus, the order of the output letters generated by a steady trace does not depend on the small deviations of the timestamps of the input signals. A TFSM $M = (S, s_{in}, G, \rho)$ is called *deterministic* iff for every pair of transitions $(s, i_1, o_1, s', (u_1, v_1], d_1)$ and $(s, i_2, o_2, s'', (u_2, v_2], d_2)$

in ρ either $i_1 \neq i_2$, or $(u_1, v_1] \cap (u_2, v_2] = \emptyset$. This requirement means that every timestamped input letter can activate no more than one transition from an arbitrary given state s . It also implies that every input timed word can activate no more than one trace in M . A deterministic TFSM is called *strictly deterministic* iff every initial trace in M which starts from the initial state s_{in} is steady. It is easy to see that TFSM, depicted in Fig. 1, is not strictly deterministic.

The Strict Determinacy Checking Problem (in what follows, SDCP) is that of checking, given a TFSM, if it is strictly deterministic. It is easy to check whether a TFSM is deterministic by considering one by one all pairs of transitions that emerge from the same state. But local means alone are not enough to check whether a given trace in a TFSM is steady. A simple criterion for steadiness of traces is presented as a Theorem below.

Let a sequence of transitions

$$(s_0, i_1, s_1, o_1, \langle u_1, v_1 \rangle, d_1), \dots, (s_{n-1}, i_n, s_n, o_n, \langle u_n, v_n \rangle, d_n)$$

be a trace tr in a TFSM M . Then the following theorem holds.

Theorem 1. A trace tr is steady iff for all pairs of integers k, m such that $1 \leq k < m \leq n$ at least one of the two inequalities $d_k - d_m \leq \sum_{j=k+1}^m u_j$ or $d_k - d_m > \sum_{j=k+1}^m v_j$ holds.

Proof. (\Rightarrow) Suppose that there exists a pair k, m such that $1 \leq k < m \leq n$, and a double inequality holds:

$$\sum_{j=k+1}^m u_j < d_k - d_m \leq \sum_{j=k+1}^m v_j.$$

Then we use two positive numbers $r = d_k - d_m - \sum_{j=k+1}^m u_j$ and $\varepsilon = \frac{r}{n}$ and consider a behaviour of a TFSM M in the input timed words

$$\alpha' = (i_1, v_1), \dots, (i_k, v_k), (i_{k+1}, u_{k+1} + \varepsilon), \dots, (i_m, u_m + \varepsilon), \\ \alpha'' = (i_1, v_1), \dots, (i_k, v_k), (i_{k+1}, v_{k+1}), \dots, (i_m, v_m).$$

It is easy to see that both words activate tr .

The trace tr converts the timed input word α_1 to the timed output word

$$conv(tr, \alpha') = \dots, (o_m, T'_m), \dots, (o_k, T'_k), \dots$$

such that $T'_m = \sum_{j=1}^k v_j + \sum_{j=k+1}^m (u_j + \varepsilon) + d_m$, and $T'_k = \sum_{j=1}^k v_j + d_k$. In this timed output word, the output letter o_k follows the output letter o_m since

$$T'_k - T'_m = d_k - d_m - \sum_{j=k+1}^m u_j + (m-k)\varepsilon = r - \frac{r(m-k)}{n} > 0.$$

Hence, $resp(tr, \alpha') = \dots, o_m, \dots, o_k, \dots$

On the other hand, the trace tr converts the timed input word α'' to the timed output word

$$conv(tr, \alpha'') = \dots, (o_k, T''_k), \dots, (o_m, T''_m), \dots$$

such that $T_k'' = \sum_{j=1}^k v_j + d_k$ and $T_m'' = \sum_{j=1}^m v_j + d_m$. In this timed output word the output letter o_m follows the output letter o_k since

$$T_m'' - T_k'' = d_m - d_k = \sum_{j=k+1}^m v_j \geq 0$$

Therefore, $\text{resp}(tr, \alpha'') = \dots, o_k, \dots, o_m, \dots$.

Thus, we got evidence that the trace tr is not steady.

(\Leftarrow) Suppose that the trace tr is not steady. Then there exists a pair of timed input words $\alpha' = (i_1, t_1'), \dots, (i_n, t_n')$ and $\alpha'' = (i_1, t_1''), \dots, (i_n, t_n'')$ such that both words activate the trace tr and $\text{resp}(tr, \alpha') \neq \text{resp}(tr, \alpha'')$. Consequently, there exists a pair of output letters o_m and o_k such that

$$\begin{aligned} \text{conv}(tr, \alpha') &= \dots, (o_k, T_k'), \dots, (o_m, T_m'), \dots \\ \text{conv}(tr, \alpha'') &= \dots, (o_m, T_m''), \dots, (o_k, T_k''), \dots \end{aligned}$$

Such permutation of output letters is possible iff the following inequalities hold

$$\begin{aligned} t_k' + d_k &= T_k' < T_m' = t_m' + d_m, \\ t_k'' + d_k &= T_k'' > T_m'' = t_m'' + d_m. \end{aligned}$$

But since both input timed words α' and α'' activate tr , we have the following chain of inequalities:

$$\sum_{j=k+1}^m u_j < T_m'' - T_k'' < d_k - d_m < T_m' - T_k' \leq \sum_{j=k+1}^m v_j.$$

Thus, if tr is not steady then there exists a pair of integers such that $1 \leq k < m \leq n$ and

$$\sum_{j=k+1}^m u_j < d_k - d_m \leq \sum_{j=k+1}^m v_j$$

holds. \square

Now, having the criterion for steadiness of traces, we can give a solution to SDCP for TFSMs. Let TFSM $M = (S, s_{in}, G, \rho)$ be a deterministic TFSM. Denote by u_{min} the greatest lower bound of all left boundaries used in the time guards of M . In our model of TFSM $u_{min} > 0$. Let d_{min} and d_{max} be the minimum and the maximum output delays occurred in the transitions of M . A theorem below gives necessary and sufficient conditions for the behaviour of M to be strictly deterministic.

Theorem 2. *A deterministic TFSM M is strictly deterministic iff all its traces of length p , where $p = \lceil \frac{d_{max} - d_{min}}{u_{min}} \rceil$, are steady.*

Proof. The necessity of conditions is obvious.

We prove the sufficiency of conditions by contradiction. Suppose that all traces of length less or equal p are steady but TFSM M is not. Then there exists such a trace tr in M which is not steady. Then, by Theorem 1, this trace is a sequence of transitions $(s_{j-1}, i_j, s_j, b_j, (u_j, v_j], d_j), 1 \leq j \leq n$, such that

for some pair of integers m and k , where $1 \leq k < m \leq n$, two inequalities

$$\sum_{j=k+1}^m u_j \leq d_k - d_m \leq \sum_{j=k+1}^m v_j$$

hold. It should be noticed, that, by the same Theorem 1, the trace tr' which includes only the transitions $(s_{j-1}, i_j, s_j, b_j, (u_j, v_j], d_j), m \leq j \leq k$, is not steady as well. Hence, $m - k > p$, and we have the following sequence of inequalities

$$d_{max} - d_{min} \geq d_m - d_k \geq \sum_{j=k+1}^m u_j > p * u_{min}$$

which contradicts our choice of $p = \lceil \frac{d_{max} - d_{min}}{u_{min}} \rceil$. \square

As it follows from Theorems 1 and 2, to guarantee that a given TFSM $M = (S, s_{in}, G, \rho)$ is strictly deterministic it is sufficient to consider all traces $(s_0, a_1, b_1, s_1, (u_1, v_1], d_1), \dots, (s_{n-1}, a_n, b_n, s_n, (u_n, v_n], d_n)$ in M , whose length n does not exceed the value $p = \lceil \frac{d_{max} - d_{min}}{u_{min}} \rceil$ defined in Theorem 2, and for every such trace check that one of the inequalities $d_1 - d_n < \sum_{j=2}^n u_j$ or $d_1 - d_n > \sum_{j=2}^n v_j$ holds. Thus, we arrive at

Corollary 1. *Strict Determinacy Checking Problem for TFSMs is decidable.*

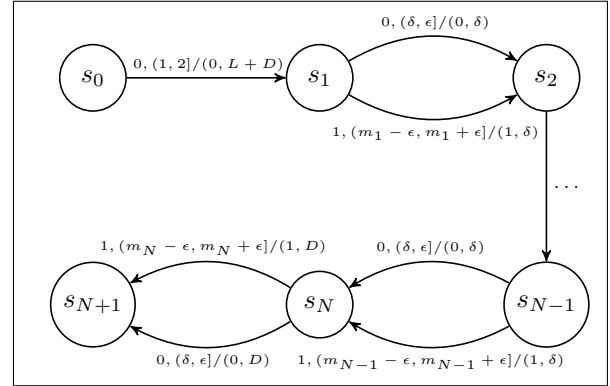


Fig.2 TFSM M

IV. STRICT DETERMINACY CHECKING PROBLEM FOR TFSMS IS CO-NP-HARD

Clearly, the decision procedure, based on Theorem 2, is time consuming since p may be exponential of the size of M and the number of traces of length p in TFSM M is exponential of p . In this section we show that such an exhaustive search can hardly be avoided because SDCP for improved version of TFSMs is co-NP-hard.

We are aimed to show that the complement of SDCP is NP-hard. To this end we consider the Subset-Sum Problem (see [7]) which is known to be NP-complete and demonstrate that this problem can be polynomially reduced to the complement of SDCP for TFSMs.

The Subset-Sum Problem (SSP) is that of checking, given a set of integers Q and an integer L , whether there is any subset $Q', Q' \subseteq Q$, such that the sum of all its elements is equal to L . More formally, the variant of the SSP we are interested in is defined as follows. Let $Q = m_1, m_2, \dots, m_N$ be a sequence of positive integers, and L be also a positive integer. A solution to (Q, L) -instance of SSP is a binary tuple $z = \langle \sigma_1, \sigma_2, \dots, \sigma_N \rangle$ such that $\sum_{j=1}^N \sigma_j m_j = L$. In [7] it was proved that the problem of checking the existence of a solution to a given (Q, L) -instance of SSP is NP-complete.

Now, given a (Q, L) -instance of SSP, we show how to build a deterministic TFSM $M_{Q,L}$ such that it has an initial trace which is *not* strictly determined iff this instance of SSP has a solution. Let $D = \sum_{j=1}^N m_j$, and ε and δ be positive rational numbers such that $\varepsilon = o(1/N^2)$ and $\delta = o(\varepsilon/N^2)$. Consider a TFSM depicted in Fig. 2. This machine operates over alphabets $I = O = \{0, 1\}$. It has $N + 2$ states $s_0, s_1, \dots, s_N, s_{N+1}$. The only transition $(s_0, 0, 0, s_1, (1, 2], L + D)$ leads from the initial state s_0 to s_1 . From each state $s_j, 1 \leq j < N$, two transitions $(s_j, 1, 1, s_{j+1}, (m_j - \varepsilon, m_j + \varepsilon], \delta)$ and $(s_j, 0, 0, s_{j+1}, (\delta, \varepsilon], \delta)$ lead to the state s_{j+1} . The state s_N is different: two transitions $(s_N, 1, 1, s_{N+1}, (m_N - \varepsilon, m_N + \varepsilon], D)$ and $(s_N, 0, 0, s_{N+1}, (\delta, \varepsilon], D)$ lead this state to s_{N+1} .

First, we make some observations.

- 1) Since all transitions outgoing from the states $s_j, 1 \leq j < N$, have the same delay δ , every trace from a state s_k to a state s_ℓ , where $0 < k < \ell \leq N$, is strictly deterministic.
- 2) Since $\delta = o(1/N^4)$ and $0 < \varepsilon = o(1/N^2)$, for every $k, 1 \leq k \leq N$, and a binary tuple $z = \langle \sigma_k, \sigma_{k+1}, \dots, \sigma_N \rangle$ the inequalities

$$\delta - D < 0 < N\delta \leq \sum_{j=k+1}^N (\sigma_j(m_j - \varepsilon) + (1 - \sigma_j)\delta)$$

hold. By Theorem 1, this implies that every trace from a state $s_k, 1 \leq k \leq N$, to the state s_{N+1} is strictly deterministic.

- 3) For the same reason the inequalities

$$D + L - \delta > \sum_{j=1}^k m_j + k\varepsilon = \sum_{j=1}^k (\sigma_j(m_j + \varepsilon) + (1 - \sigma_j)\varepsilon)$$

hold for every $k, 1 \leq k < N$, and a binary tuple $z = \langle \sigma_1, \sigma_2, \dots, \sigma_k \rangle$. By Theorem 1, this guarantees that every initial trace leading to a state $s_k, 1 \leq k \leq N$ is strictly deterministic.

As for the initial traces that lead to the state s_{N+1} , due to our choice of ε and δ , we can trust the following chain of reasoning. By definition, a (Q, L) -instance of SSP has a solution $z = \langle \sigma_1, \sigma_2, \dots, \sigma_N \rangle$ iff $\sum_{j=1}^N \sigma_j m_j = L$. The latter is possible iff two following inequalities hold:

$$\sum_{j=1}^N \sigma_j m_j - \varepsilon + N\delta < L < \sum_{j=1}^N \sigma_j(m_j) + N\varepsilon \quad (1)$$

By taking into account the relationships below

$$\begin{aligned} \sum_{j=1}^N (\sigma_j(m_j - \varepsilon) + (1 - \sigma_j)\delta) &< \sum_{j=1}^N \sigma_j m_j - \varepsilon + N\delta \\ \sum_{j=1}^N \sigma_j(m_j) + N\varepsilon &= \sum_{j=1}^N (\sigma_j(m_j + \varepsilon) + (1 - \sigma_j)\varepsilon), \end{aligned}$$

we can conclude that (1) holds iff another pair of inequalities hold:

$$\sum_{j=1}^N (\sigma_j(m_j - \varepsilon) + (1 - \sigma_j)\delta) < L < \sum_{j=1}^N (\sigma_j(m_j + \varepsilon) + (1 - \sigma_j)\varepsilon)$$

But in the context of observations 1) – 3) above, the latter inequalities, as it follows from Theorem 1, provide the necessary and sufficient conditions that the initial trace in TFSM $M_{Q,L}$ activated by the input word $z = \langle \sigma_1, \sigma_2, \dots, \sigma_N \rangle$ is not strictly deterministic.

Thus, a (Q, L) -instance of SSP has a solution iff TFSM $M_{Q,L}$ is not strictly deterministic.

The considerations above bring us to

Theorem 3. *SDCP for TFSMs is co-NP-hard.*

V. CONCLUSION

The main contributions of this paper are

- 1) the development of a modified version of TFSM which, in our opinion, provides a more adequate model of real-time computing systems;
- 2) the introduction of the notion of strict deterministic behaviour of TFSM and setting up the Strict Determinacy Checking Problem (SDCP) for a modified version of TFSMs;
- 3) the establishing of an effectively verifiable criterion for the strict determinacy property of TFSMs;
- 4) the proving that SDCP for TFSMs is co-NP-hard.

However, some problems concerning strict deterministic behaviour of TFSMs still remain open. They will be topics for our further research.

1. In Sections III and IV it was shown that SDCP for TFSMs is co-NP-hard and in the worst case it can be solved in double exponential time by means of a naive exhaustive searching algorithm based on Theorems 1 and 2. We think that this complexity upper bound estimate is too much high. The question arises, for what complexity class \mathcal{C} SDCP for TFSMs is a \mathcal{C} -complete problem. By some indications we assume that SDCP for TFSMs is PSPACE-complete problem.

2. As it can be seen from the proof of Theorem 3, SDCP for TFSMs is intractable only if timed parameters of transitions (time guards and delays) depend on the number of states in TFSM. But this is not a typical phenomenon in real-time systems since in practice the performance of individual components of a system does not depend on the size of the system. Therefore, it is reasonable to confine ourselves to considering only such TFSMs, in which the time guards and the delays are chosen from some fixed finite set. As it follows from Theorem 2, for this class of TFSMs SDCP is decidable

in polynomial time. One may wonder what is the degree of such a polynomial, or, in other words, how efficiently the strict determinacy property can be checked for TFMSs corresponded to real systems.

3. In the model of TFMS besides the usual transitions there are also possible timeout transitions. A timeout transition fires when a timestamped input letter (i, t) can not activate any usual transition from a current state. In [5] it was shown that in some cases such timeout transitions can not be replaced by any combination of ordinary transitions. In the future we are going to study how SDCP can be solved for TFMSs with timeouts.

The authors of the article express their deep gratitude to V.V. Podymov and the anonymous reviewers for their valuable comments and advice on improving the article.

This work was supported by the Russian Foundation for Basic Research, Grant N 18-01-00854.

REFERENCES

- [1] Alur R., Dill D. *A Theory of Timed Automata*. Theoretical Computer Science, 1994, vol. 126, p. 183-235.
- [2] Alur R., Madhusudan P. *Decision Problems for Timed Automata: A Survey*. Proceedings of the 4-th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'04), 2004, p. 1-24.
- [3] Alur R., Fix L., Henzinger Th. A. *A Determinizable Class of Timed Automata*. Proceedings of the 6-th International Conference on Computer Aided Verification (CAV94), 1994, p. 1-13.
- [4] Baier C., Bertrand N., Bouyer P., Brihaye T. *When are Timed Automata Determinizable?* Proceedings of the 36-th International Colloquium on Automata, Languages, and Programming (ICALP 2009), 2009, p. 43-54.
- [5] Bresolin D., El-Fakih K., Villa T., Yevtushenko N. *Deterministic Timed Finite State Machines: Equivalence Checking and Expressive Power*. International Conference GANDALF, 2014, p. 203-216.
- [6] Cardell-Oliver R. *Conformance Tests for Real-Time Systems with Timed Automata Specifications*. Formal Aspects of Computing, 12(5), 2000, p. 350371.
- [7] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. "35.5: The subset-sum problem" // Introduction to Algorithms (2-nd ed.), 2001.
- [8] Finkel O. *Undecidable Problems about Timed Automata*. Proceedings of 4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS06), 2006, p. 187-199.
- [9] Fletcher J. G., Watson R. W. *Mechanism for Reliable Timer-Based Protocol*. Computer Networks, 1978, vol. 2, p. 271-290.
- [10] Merayo M.G., Nuñez M., Rodríguez I. *Formal Testing from Timed Finite State Machines*. Computer Networks, 2008, vol. 52, No 2, p. 432-460.
- [11] Ouaknine J., Worrell J. *On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap*. Proceedings of the 19-th Annual Symposium on Logic in Computer Science (LICS04), 2004, p. 54-63.
- [12] Suman P.V., Pandya P.K., Krishna S.N., Manasa L. *Timed Automata with Integer Resets: Language Inclusion and Expressiveness*. Proceedings of the 6-th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS08), 2008, p. 7892.
- [13] Tvardovskii A., Yevtushenko N. *Minimizing Timed Finite State Machines* // Tomsk State University Journal of Control and Computer Science, No 4 (29), 2014, p. 77-83.
- [14] Tvardovskii A., Yevtushenko N. *Minimizing Finite State Machines with Time Guards and Timeouts* // Proceedings of ISP RAS, vol. 29, Issue 4, 2017, p. 139-154.
- [15] Zhigulin M., Yevtushenko N., Maag S., Cavalli A. *FSM-Based Test Derivation Strategies for Systems with Timeouts*. Proceedings of the 11-th International Conference on Quality Software, 2011, p. 141-149.

Deriving adaptive distinguishing sequences for Finite State Machines

Aleksandr Tvardovskii
National Research Tomsk State University
Tomsk, Russia
tvardal@mail.ru

Nina Yevtushenko
Institute for System Programming of the Russian Academy of
Sciences
Moscow, Russia
evtushenko@ispras.ru

Abstract—Distinguishing sequences (DS) are used in FSM (Finite State Machines) based testing for state identification and can significantly reduce the size of a returned complete test. However, such sequences not always exist for deterministic and nondeterministic FSMs and are rather long when existing. Adaptive DS are known to exist more often and be much shorter that makes them attractive for test derivation. In this paper, we investigate properties of adaptive distinguishing sequences and propose an approach for optimizing the procedure of adaptive DS derivation.

Keywords—Finite State Machine (FSM); test case; adaptive distinguishing sequence

I. INTRODUCTION

Finite State Machines (FSMs [1]) are widely used for deriving tests with the guaranteed fault coverage for reactive discrete event systems. The well-known are the W-method [2] and many its derivatives for deterministic and nondeterministic FSMs [see for example, 3 and 4]. In FSM based test derivation, the specification behavior and the behavior of an implementation under test (IUT) are described by FSMs and a test suite is derived that detects each non-conforming implementation of a given fault domain. The well-known conformance relations are the equivalence [2] and reduction relations [3]. The equivalence means that an IUT has to have the same behavior as the specification FSM (trace equivalence) while the reduction relation is used when the behavior of an IUT is allowed to be contained in the specification behavior (trace containment).

FSM based test derivation methods rely on the state identification sequences in the specification FSM, namely, on distinguishing sequences (DS) that can be preset or adaptive [5]. Preset input sequences are derived before starting the test derivation procedure, while for adaptive sequences, the next input depends on the outputs produced for the previous inputs. Adaptive sequences are represented by a tree or an acyclic FSM [3] called a *test case*. It is also known that usually a test suite can be shorter if the specification FSM has a sequence which distinguishes every two states [3, 10] but such a distinguishing sequence does not always exist. Moreover, the length of a distinguishing sequence can exponentially depend on the number of states of the specification FSM. Adaptive distinguishing sequences exist more often than the preset and are usually shorter, thus, adaptive distinguishing sequences can be preferable for test derivation.

When deriving adaptive distinguishing sequences the authors consider a successor or a spanning tree. However, based on the above trees for nondeterministic FSMs, there are no necessary and sufficient conditions when an adaptive distinguishing sequence exists. The authors of [9] propose another formal approach that is based on deriving an appropriate distinguishing FSM and establish the necessary and sufficient conditions for the existence of an adaptive distinguishing sequence. In this paper, we consider the problem of deriving adaptive distinguishing sequences for a complete possibly nondeterministic FSM and propose an approach for optimizing the procedure for deriving an adaptive distinguishing sequence based on a distinguishing FSM. The performed experiments with randomly generated nondeterministic FSMs demonstrate that in many cases, an optimized distinguishing machine is simpler and the existence / nonexistence of adaptive distinguishing sequences can be faster determined.

The rest of the paper has the following structure. Section II contains the preliminaries. The procedure for deriving an adaptive distinguishing sequence based on a distinguishing FSM and its optimization are presented in Section III. Section IV contains experimental results and Section V concludes the paper.

II. PRELIMINARIES

In this section, we introduce necessary definitions and notations which are mainly taken from the papers [6, 9].

A. Finite State Machines

A *finite state machine* (FSM), or simply a *machine*, is a 5-tuple $S = \langle S, I, O, h_S, s_0 \rangle$ where S is a finite non-empty set of states with the designated initial state s_0 , I and O are finite input and output alphabets, and $h_S \subseteq S \times I \times O \times S$ is a *transition relation*. FSM S is *nondeterministic* if for some pair $(s, i) \in S \times I$, there can exist several pairs $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$; otherwise, the FSM is *deterministic*. FSM S is *complete* if for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$; otherwise, the FSM is *partial*. FSM S is *observable* if for every two transitions $(s, i, o, s_1), (s, i, o, s_2) \in h_S$ it holds that $s_1 = s_2$. In the following, we consider complete observable possibly nondeterministic FSMs if the contrary is not directly stated. An example of a complete nondeterministic FSM with $S = \{0, 1, 2, 3\}$, $I = \{0, 1, 2\}$, $O = \{0, 1\}$ is shown in Fig. 1.

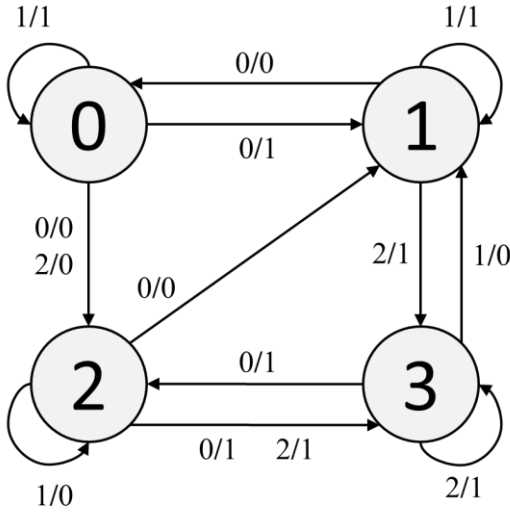


Fig 1. Complete nondeterministic FSM S

Given an input/output pair io and a state s of a complete observable FSM S , state s' is the io -successor of state s of FSM S if $(s, i, o, s') \in h_S$. The io -successor of state s not necessary exists and in this case, we say that the io -successor of state s is the empty set. A *trace* of FSM S at state s is a sequence of input/output pairs which label consecutive transitions starting from state s , $tr = i_1/o_1 \dots i_l/o_l$. A sequence $i_1 \dots i_l$ is an *input* sequence of the trace, a sequence $o_1 \dots o_l$ is an *output* sequence.

FSM S is *merging-free* [7] if for every two states s_1 and s_2 and any input i it holds that if $(s_1, i, o, s'_1), (s_2, i, o, s'_2) \in h_S$, then $s'_1 \neq s'_2$.

B. Test Case definition

An input sequence α is *adaptive* if the next input depends on the output to the previous one. Such an input sequence can be represented by a special FSM called a test case [8].

Given an input alphabet I and an output alphabet O , a *test case* $TC(I, O)$ (over I and O) is an initially connected observable FSM $T = \langle T, I, O, h_T, t_0 \rangle$ with an acyclic transition graph such that at each state either only one input with all possible outputs is defined or there are no outgoing transitions. Given a complete FSM S over alphabets I and O , a test case $TC(I, O)$ represents an adaptive experiment with the FSM S . If $|I| > 1$ then a test case is a partial FSM. A state $t \in T$ is a *deadlock* state of the FSM T if there are no defined inputs at this state. The *length (height)* of the test case T is defined as the length of a longest trace from the initial state to a deadlock state of T and it specifies the length of the longest input sequence that can be applied to an FSM S during the adaptive experiment.

A test case T is a *distinguishing* test case (DTC) for an FSM S if for every trace γ of T from the initial state to a deadlock state, γ is a trace at most at one state of S . Sometimes, a distinguishing test case is called an *adaptive distinguishing*

sequence. A distinguishing test case D for a FSM S in Figure 1 is shown in Figure 2.

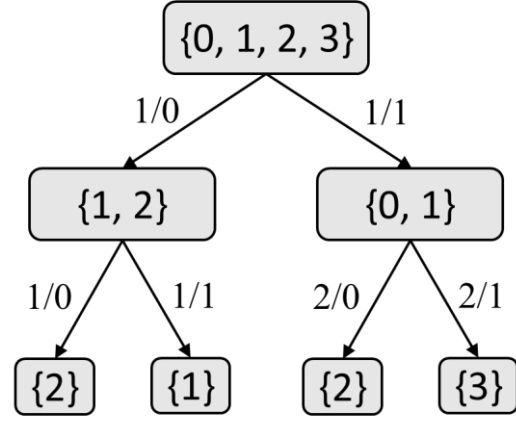


Fig 2. DTC D for a FSM S

Given a DTC D for FSM S , an adaptive distinguishing sequence defined by DTC D is applied in the following way. If input i_1 is defined at the initial state d_0 of D then first input i_1 is applied to FSM S and DTC D moves to the $i_1 o$ -successor d_1 of state d_0 if the output o is the response of S to the input i_1 . The next input to apply is the input defined at state d_1 , etc. The procedure terminates when a deadlock state is reached. The corresponding trace allows to determine a state of the FSM S before the experiment. For example, consider a DTC in Figure 2. At the first step an input 1 that is a defined input at the initial state $\{0, 1, 2, 3\}$ is applied. If an output 1 is produced by the FSM under experiment then the DTC moves to state $\{0, 1\}$ and the defined input 2 at this state is applied. Let the FSM produce the output 2 and consider a trace $1/1 \ 2/1$ that is a trace from the initial state to a deadlock state in Figure 2. By direct inspection, one can assure that FSM S in Figure 1 has such a trace only at state 1.

III. TEST CASE DERIVATION

The method for deriving a distinguishing test case for a nondeterministic FSM based on a distinguishing machine is described in [9]. Given a complete observable nondeterministic FSM $S = (S, I, O, h_S)$, a distinguishing FSM S_{dist} is derived in the following way. The set of inputs (outputs) S_{dist} coincides with the set of inputs (outputs) of S , while states S_{dist} are defined over the set of all non-empty and non-singleton subsets of states of S and the distinguishing machine has a special state F . The process of constructing transitions of S_{dist} starts at the initial state which corresponds to the set of all states of FSM S and the machine S_{dist} is a minimal machine that can be constructed using the following rules.

Given a state b of S_{dist} , let b be a non-empty and non-singleton subset of states of S .

- 1) There is transition (b, i, o, b') in S_{dist} if and only if b' is not a singleton, for every $o' \in O$, the non-empty io' -successors of every two different states of the set b do not coincide, and b' is the non-empty io -successor of

the set b . In this case, state b' is added to the set of states of S_{dist} .

- 2) There is a transition (b, i, o, F) if and only if there exists $o' \in O$ such that the non-empty io' -successors of two different states of the set b coincide. In this case, state F is added to the set of states of S_{dist} .
- 3) A transition from state b under input i in S_{dist} is labeled as an undefined transition if and only if for every $o' \in O$, the non-empty io' -successors of every two different states of the set b do not coincide and each io -successor of the set b is singleton.

At the state F , there is a transition (F, i, o, F) for every i/o pair, $i \in I, o \in O$.

By definition of a distinguishing FSM S_{dist} , an input i is an *undefined* input at state b if there are no transitions from state b under i , i.e., if for every $o' \in O$, the non-empty io' -successors of every two different states of the set b do not coincide and each io -successor of the set b is singleton. In order to check if FSM S has a DTC, states with undefined inputs are iteratively removed from S_{dist} with all incoming transitions until either there are no undefined inputs in S_{dist} or the initial state of S_{dist} has an undefined input. When deleting a state b with undefined inputs, we denote $UN(b) = i$ for some undefined input i . The initial state has an undefined input if and only if FSM S has a DTC D that can be derived from S_{dist} by using saved undefined inputs when removing states, i.e., the specification FSM S has a DTC if and only if the machine S_{dist} has no complete submachine. In this case, the initial state d_0 of D corresponds to the initial state b_0 of S_{dist} . The input $UN(b_0)$ is the only defined input at state d_0 of D and D has a transition $(d_0, UN(b_0), o, d)$ if and only if FSM S_{dist} had a transition $(b_0, UN(b_0), o, b)$ before removing of states. If for some $o \in O$, there is no transition $(b_0, UN(b_0), o, b)$ in S_{dist} , then transition $(d_0, UN(b_0), o, DL)$ is added to FSM D where DL is the deadlock state. Transitions for every state d of DTC D are constructed in a similar way. For the FSM S in Figure 1, the corresponding distinguishing machine S_{dist} is shown in Figure 3; this machine has nine states and a DTC of height three can be constructed by the iterative removal of states with undefined transitions. In fact, the above contains the informal proof how to check whether FSM S has a DTC using machine S_{dist} .

According to the results in [9], the length of a DTC can reach $2^{n-1} - 1$ for a complete observable FSM with n states and for the class of such FSMs, it can happen that each non-empty and non-singleton subset of S is a state of S_{dist} , and thus, there can occur the state explosion problem when constructing a corresponding S_{dist} . For this reason, we propose to limit the construction with states which are reachable from the initial state by an input sequence limited by the length $L > 0$, i.e., to construct S_{dist}^L , since according to the reference to experimental results in [9], the length of a DTC for randomly generated machines with up to 100 states does not exceed 15.

In other words, when constructing the distinguishing machine S_{dist}^L , we propose to limit the set of states only with states which can be reached from the initial state by a trace of length up to L . All the transitions at a state reachable only via a trace of length L are directed to the designated state F and state

F has a loop for each i/o pair, $i \in I, o \in O$. Similar to the use of the machine S_{dist} , the following theorem can be proven.

Theorem 1. The specification FSM S has a DTC of length up to L if and only if the machine S_{dist}^L has no complete submachine.

In this case, the same iterative procedure of deleting states with undefined inputs can be applied for checking the existence of a DTC of height up to L as well as for deriving a DTC (if it exists). Before writing the corresponding procedure we illustrate the above optimization for a distinguishing machine in Figure 1. Already after construction a machine in Figure 3 for traces of length up to two we can see that there is a DTC of length two (Figure 2) in the specification FSM.

For the distinguishing FSM in Figure 3, grey states are states of FSM S_{dist}^2 . This distinguishing machine S_{dist}^2 has no complete submachine and a corresponding DTC of height two is shown in Figure 2.

Thus, the following procedure for deriving a DTC for a complete observable possibly nondeterministic FSM can be proposed.

Optimized procedure for deriving DTC

Input: a complete observable FSM S , integer $L > 0$

Output: A DTC of length up to L for FSM S or the message 'FSM S has no DTC of length up to L '

$l := 1$, the set of undefined inputs $UN := \emptyset$, distinguishing FSM S_{dist}^0 with the only initial state; $Q = S_{dist}^0$;

Step 1. Add to S_{dist}^{l-1} states which are reachable by a trace of length l , i.e., derive a distinguishing FSM S_{dist}^l and copy all new transitions (states) to FSM Q .

If $S_{dist}^l = S_{dist}^{l-1}$ **then** output the message 'FSM S has no DTC' and **END** the procedure.

Else Step 2.

Step 2. Iteratively delete from S_{dist}^l each state b that has an undefined input with its incoming transitions, select an undefined input i and denote $UN(b) = i$.

If the initial state has an undefined transition **then** Step 3

Else

If $l + 1 > L$ **then** output the message 'FSM S has no DTC of length up to L ' and **END** the procedure

Else $l = l + 1$ and Step 1.

Step 3. Derive a DTC using the set UN of undefined inputs and the distinguishing FSM Q which coincides with the FSM S_{dist}^l ;

END

Theorem 2. Given a complete observable possibly nondeterministic FSM S , an above optimized procedure returns a shortest DTC D for FSM S if there exists a DTC of length up to L .

Indeed, due to Theorem 1, there exists a DTC of length up to k if and only if the FSM S_{dist}^k has no complete submachine, i.e., at the k -th iteration the initial state of the distinguishing machine will have an undefined input. As we start with $k = 1$ and the procedure terminates once there is a DTC of height $k < L + 1$ (Step 2), the procedure returns a shortest DTC of height up to L if such a DTC exists.

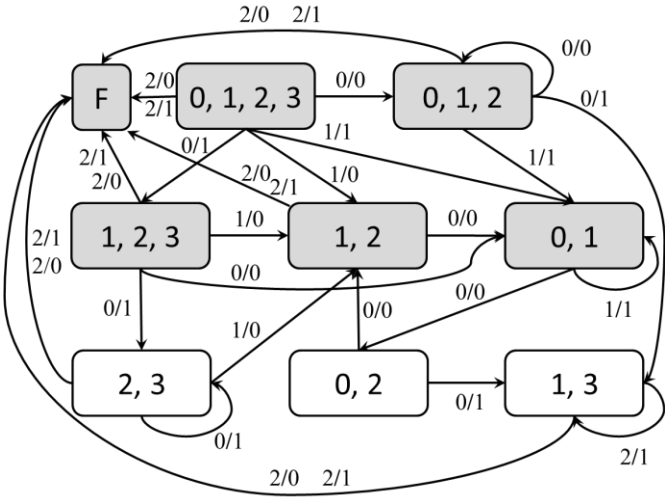


Fig. 3. Distinguishing FSM for a FSM S in Fig. 1

As mentioned above, in Figure 3, only grey states and the designated state F are left in the distinguishing machine $S^{2_{dist}}$ that correspondingly has six states; all the transitions from these states to non-grey states are directed to the designated state F . By direct inspection, one can assure that this machine has no complete submachine, since after two iterations the initial state has an undefined input. Correspondingly, FSM S in Figure 1 has a DTC of length 2 (Figure 2).

The appropriate length L can be determined after a number of experiments. Both, original and optimized methods, have been implemented and in the next section, we present the obtained experimental results.

IV. EXPERIMENTAL RESULTS

In this section, we present experimental results on checking the existence of a DTC for deterministic and nondeterministic FSMs with a number of parameters and evaluate the DTC height when it exists. When performing the experiments, randomly generated FSMs were utilized [11].

A. Randomly generated FSMs

We first performed experiments with randomly generated nondeterministic FSMs with various numbers of transitions for every pair 'state, input'. The maximum number of transitions is further denoted nd and our experiments clearly show that a DTC rarely exists when nd is more than three (Figure 4). Correspondingly, for other experiments, $nd \leq 3$ is only considered.

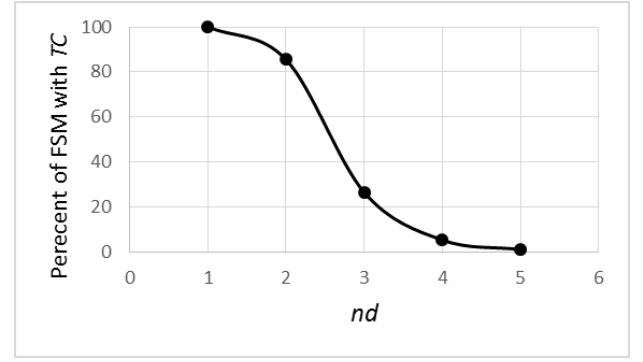


Fig. 4. Percentage of FSMs when a DTC exists with respect to the number of transitions for each pair 'state, input', $|I| = |O| = |S| = 10$

In Figure 5, the DTC length is shown depending on the number of FSM states. The lowest curve demonstrates the average length of DTC for deterministic FSMs while two upper curves are related to nondeterministic FSMs where $nd = 2$ for the middle curve and $nd = 3$ for the upper curve. FSMs with up to 30 states are considered with $|I| = |O| = 10$.

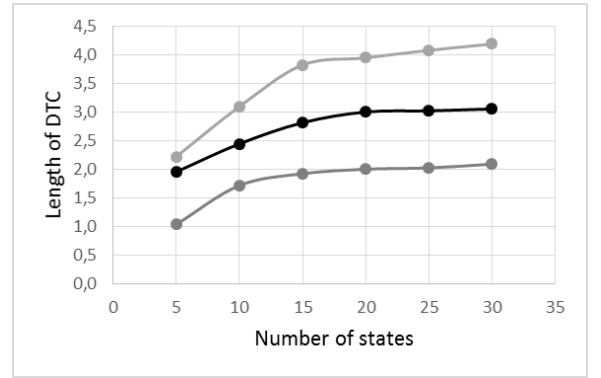


Fig. 5. DTC length for deterministic and nondeterministic FSMs

As we can see, according to the experimental results, randomly generated FSMs either have a rather short DTC or have no DTC at all. The latter is due to the fact that randomly generated FSMs have a big number of merging transitions, especially for $nd \geq 3$. Respectively, corresponding distinguishing FSMs are not large and an optimized approach based on the distinguishing machine $S^{L_{dist}}$ proposed in Section III did not significantly overcome the approach based on the FSM S_{dist} . In particular, for FSMs with $|S| = 30$, $|I| = |O| = 10$ and $nd = 2$, the implementation of an optimized approach was twice faster on average. The runtime of the DTC derivation by both approaches for FSMs with $|I| = |O| = 10$ and $nd = 3$ is demonstrated below. In Figure 6, the lower curve corresponds to the runtime of an optimized procedure while the upper curve corresponds to the approach based on the machine S_{dist} .

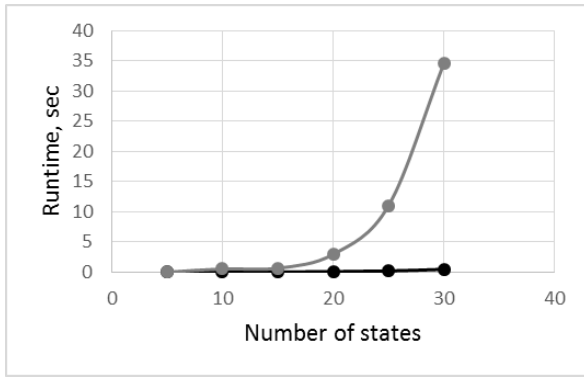


Fig 6. The runtime of original and optimized procedures for DTC derivation ($|I| = |O| = 10$ and $nd = 3$)

Note, that similar curves can be derived for memory used by the original and optimized procedures. However the dependence on the number of distinguishing FSM states is different. For FSMs with 30 states, $|I| = |O| = 10$ and $nd = 3$, on average, FSMs S_{dist}^4 and S_{dist} have 847 and 1756 states respectively. The obtained results clearly show that the increase of the nondeterminism degree leads to significant resource increasing and respectively, the DTC derivation by an optimized procedure becomes preferable.

In the next subsection, we consider a proper class of so-called merging-free FSMs which is actively investigated and for which a DTC exists more often.

B. Merging-free FSMs

A merging-free FSM has a DTC if and only if each pair of states has a DTC [7]. FSMs of this class can be used as formal models of real systems and the check of the existence of a DTC is simpler for such FSMs. In particular, randomly generated merging-free FSMs have a DTC more often and experiments were conducted for FSMs with larger number of states. For merging-free FSMs, a distinguishing machine has no transitions to the designated state F and due to the absence of merging transitions, distinguishing FSMs are significantly bigger than those for arbitrary FSMs with merging.

For merging-free FSMs with $nd = 3$, distinguishing FSMs are rather big and below the runtime evaluation is performed only for $nd \leq 2$.

Since the runtime and memory for constructing a distinguishing FSM S_{dist} increase for merging-free FSMs, a proposed optimization for using S_{dist}^L for the test case derivation becomes more efficient. For example, given merging-free FSMs with 50 states, $|I| = |O| = 10$ and $nd = 2$ and looking for a DTC of length up to three, the number of states of FSMs S_{dist}^3 and S_{dist} are 4992 and 8202 respectively. Curves for the runtime of an original and an optimized procedure for the DTC derivation are presented below.

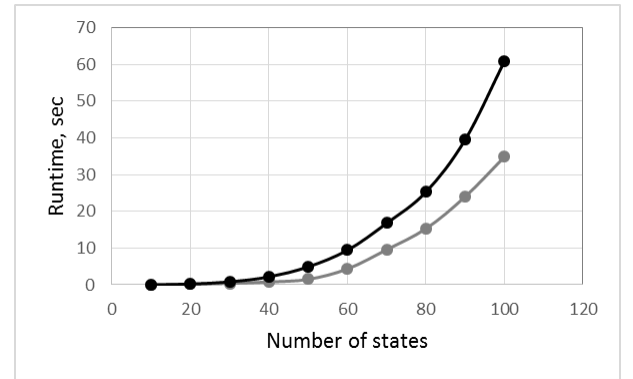


Fig 7. The runtime of an original and an optimized procedure for DTC derivation for merging-free FSMs ($|I| = |O| = 10$ and $nd = 2$)

Similar to ordinary FSMs, the optimized procedure is twice faster than the original one when $|S| < 100$ and $nd = 2$. However, the runtime is increased for merging-free FSMs compared with the common case.

The results on the DTC length evaluation almost coincide with those in Figure 5. For FSMs with 100 states, the length of a DTC (when it exists) does not exceed 5 for merging-free FSMs when $|I| = |O| = 10$ and $nd = 2$. At the same time, merging-free FSMs have DTC much more often than ordinary FSMs. For random generated merging-free FSMs with 100 states the percentage of FSMs with a DTC is almost 100% when $|I| = |O| = 10$ and $nd = 2$.

V. CONCLUSIONS

In this paper, we have investigated approaches for adaptive distinguishing sequence derivation for a complete observable possibly nondeterministic FSM and proposed an optimized procedure for deriving a distinguishing test case (DTC) that represents an adaptive distinguishing sequence. Experiments were conducted for the evaluation of the effectiveness of a proposed procedure as well as for the evaluation how often adaptive distinguishing sequences exist for nondeterministic FSMs depending on the number of different transitions for each pair 'state, input'. The experimental results show that the length of an adaptive distinguishing sequence for randomly generated FSMs does not reach the worst exponential complexity with respect to the number of FSM states. For such stress testing a corresponding FSM class has to be implemented for which the same experiments should be conducted.

REFERENCES

- [1] A. Gill. Introduction to the Theory of Finite-State Machines. 1964, 272 p.
- [2] T.S. Chow. Testing software Design Modelled by Finite-State Machines. IEEE Transactions on Software Engineering, vol. 4, issue 3, 1978, pp. 178-187.
- [3] A. Petrenko and N. Yevtushenko. Conformance Tests as Checking Experiments for Partial Nondeterministic FSM. Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), LNCS 3997, 2005, pp. 118-133.
- [4] R. Dorofeeva, K. El-Fakih, S. Maag, A.R. Cavalli, N. Yevtushenko. FSM-based conformance testing methods: a survey annotated with experimental evaluation. Information and Software Technology, 52, 2010, pp. 1286-1297.

- [5] R. Alur, C. Courcoubetis, and M. Yannakakis, "Distinguishing tests for nondeterministic and probabilistic machines", Proc. of the 27th ACM Symposium on Theory of Computing, 1995, pp. 363-372.
- [6] A. Petrenko, and N. Yevtushenko. "Adaptive testing of deterministic implementations specified by nondeterministic FSMs", Proc. of the International Conference on Testing Software and Systems, LNCS 7019, 2011, pp. 162-178.
- [7] N.Yevtushenko, N. Kushik. Nondeterministic merging-free finite state machines. Proceedings of IEEE East-West Design & Test Symposium (EWDTS), 2015, pp. 338–341.
- [8] N. Yevtushenko, K. El-Fakih, and A. Ermakov. On-the-fly construction of adaptive checking sequences for testing deterministic implementations of nondeterministic specifications, Lect. Notes Comput. Sci., 2016, vol. 9976, pp. 139–152.
- [9] K. El-Fakih, N. Yevtushenko, N. Kushik. Adaptive distinguishing test cases of nondeterministic finite state machines: test case derivation and length estimation. Formal Aspects of Computing vol. 30, issue 2, 2018, pp. 319-332.
- [10] A. Tvardovskii. Refining the Specification FSM When Deriving Test Suites w.r.t. the Reduction Relation. Lecture Notes in Computer Science (LNCS), 2017, Vol. 10533, pp. 333-339.
- [11] N. Shabaldina, M. Gromov. FSMTTest-1.0: a manual for researches. Proceedings of the 13th Intern symposium on IEEE EAST-WEST DESIGN & TEST SYMPOSIUM (EWDTS'15), 2015, pp. 216–219.

Prosega/CPN: An Extension of CPN Tools for Automata-based Analysis and System Verification

Julio César Carrasquel
Department of Computer, Control
and Management Engineering
La Sapienza University of Rome
Rome, Italy
julio.carrasquel@yahoo.com

Ana Morales
School of Computer Science
Central University of Venezuela
Caracas, Venezuela
ana.morales@ciens.ucv.ve

María Elena Villapol
School of Engineering, Computer
and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
maria.villapol@aut.ac.nz

Abstract—The combination of Coloured Petri Nets (CPNs) and Automata Theory has proved to be a successful formal technique in the modelling and verification of different distributed systems. In this context, this paper presents Prosega/CPN (*Protocol Sequence Generator and Analyzer*), an extension of CPN Tools for supporting automata-based analysis and verification. The tool implements several operations such as the generation of a minimized deterministic finite-state automaton (FSA) from a CPN's occurrence graph, language generation, and FSA comparison. The tool is intended to support a formal verification methodology of communication protocols; however, it may be used in the verification of other systems whose analysis involves the comparison of models at different levels of abstraction. An insightful use case is provided where Prosega/CPN has been used to analyze part of the IEEE 802.16 MAC connection management service specification.

Index Terms—formal methods, Coloured Petri nets, CPN Tools, finite-state automata (FSA), protocol verification

I. INTRODUCTION

The verification of distributed systems, and the assurance of their correctness is a task of utmost importance; specially in today's world where many critical services are completely supported by computer technologies. Among the solutions for system modelling and verification, Petri Nets [1] play a major role since its capability of graphically visualize systems, and for maintaining the formal rigor, so it allows to perform a convenient analysis of the behavioral properties of a system. Thus, the formalism of Petri Nets has been extended to other models in order to enrich their expressiveness and practicability. Particularly, we consider Coloured Petri Nets (CPNs) [2] where data types (*colours*) may be associated to net elements. CPN Tools [3] is a consolidated software tool for editing, simulating, and analyzing CPN models.

However, when dealing with a higher complexity of the system, it may be useful to combine the usage of different analysis techniques. This also allows the application of the best formalism or technique to different components of a system. In the context of Coloured Petri Nets, the last version of CPN Tools includes the Simulator Extensions whose development has been driven by the need of integrating CPN with other formal methods [4]. In particular, we consider the integration of CPNs and Finite-state Automata (*FSA*) which has been

proved to be useful for the validation of different protocols and communication systems [5] [6] [7].

For instance, given a CPN's occurrence graph (*OG*), the arcs through a path in the *OG* may be seen as the sequence of service primitives that a user (i.e. another system entity in a higher layer) invokes in order to request some action by a service provider. The nodes in the *OG* may be considered as changes of state in the system due to the services invocations. Finally, some nodes of the *OG* may represent halt states, meaning the termination of a specific process. Hence, the *OG* can be seen as a *FSA* which can be analyzed using well-known algorithms and theorems.

There are several tools for building, combining, optimizing, and searching Finite-state Automata. However, in order to apply them for analyzing CPNs and occurrence graphs, these ones must be converted into *FSA* specific formats (i.e. see [5] [6]). Using several tools may complicate the verification process.

Thereby, we developed a solution called Prosega/CPN (*Protocol Sequence Generator and Analyzer*). The tool aims to bridge conveniently the formalism of CPNs with Finite-state Automata, taking advantage of the Simulator Extensions feature in CPN Tools. Thus, the software provides a mechanism for transforming a CPN's occurrence graph into a minimized deterministic *FSA* as well as other operations for language generation and *FSA* comparison. Prosega/CPN has been conceived to support the protocol verification methodology proposed by Billington [8]. However, the tool may be useful to support the verification of other systems whose strategy may involve the usage of *FSAs*, or the comparison of models at different levels of abstraction; for example, business strategy and business processes.

The remainder of this paper is structured as follows. Section II introduces the literature related to our work. Section III presents some formal definitions for understanding the models managed by Prosega/CPN. Sections IV and V describe the tool functionalities and architecture respectively. Section VI describes a use case where the tool has been used to analyze part of the IEEE 802.16 MAC connection management service specification. Finally, Section VII presents the conclusions.

II. RELATED WORK

Prosega/CPN has been developed within the context of system verification through the formalism of Coloured Petri Nets (CPNs) and Finite-state Automata (*FSA*). The tool has been conveniently developed as an extension of CPN Tools [3] since it performs several operations on *FSAs* generated from a CPN model. i.e. the reduction of a CPN's occurrence graph into a *FSA*. Hence, through the development of Prosega/CPN we have been focused in three topics within the literature:

- (i) Works dealing with the development of extensions for CPN Tools [4] [9] [10] [11].
- (ii) Tools and other solutions for the analysis and manipulation of *FSA* [12] [13] [14] [15] [16].
- (iii) Works proposing a system verification methodology using CPNs and *FSA*, and the use cases in which it has been applied [5] [6] [7] [8] [17], and other scenarios where both formalisms have been used together [18] [19] [20].

CPN tools has a history for communicating with external solutions; its architecture provides a set of communication primitives for connecting external software to the CPN simulator engine. As an initial effort it was developed Comms/CPN [9], a library for Java and C/C++ which makes it possible for CPN Tools to communicate based on TCP/IP with external application and processes; the BRITNeY Suite [10] is other solution which provides model visualizations in an external tool, and more recently Access/CPN [11] that provides a channel to interact with the CPN Tools simulator engine from external Java programs. However, while these previous tools have made it easy to *interact* with CPN Tools, they have not made it possible to *extend* the software. Thereby, it was developed the Simulator Extensions [4] feature included in the last version of CPN Tools. This component provides a mechanism for adding new functionalities within the CPN Tools Graphical User Interface (GUI), thereby allowing to integrate other related formalisms with CPN models; as a result, it has been possible to handle other models in the tool such as low-level Place/Transition nets, Declare models, and drawing message sequence charts from model executions [4].

On the other hand, Finite-state Automata (*FSA*) have been used in a much wider spectrum of fields than CPNs; as an important tool for *FSA* manipulation we highlight the FSM Library from AT&T Labs [12] which is a collection of Unix software tools for creating and manipulating finite-state machines. Despite the library is quite general purpose, it was designed for speech processing applications such as speech recognition/synthesis; FSM Library was used as well in previous works regarding the verification of communication systems based on CPNs and automata [5] [6]. Some of the researchers of the AT&T FSM project developed later an enhanced version called OpenFST [13] which is an open-source alternative that also allows to construct finite-state transducers, and it provides a C++ template library. Within the range of tool solutions for *FSA* manipulation we may

also find Foma [14], the FAdo project [15] and the specialized pedagogical tool JFLAP [16] among many others.

Bridging CPNs and *FSA* may be useful for verification of systems of very high complexity. In particular, Billington [8] proposed a CPN and *FSA* approach for the verification communication systems that has proven to be successful; namely, in the verification of the Resource Reservation Protocol (RSVP) [5], the Wireless Application Protocol (WAP) [6], the Transmission Control Protocol (TCP) [7], and the Internet Open Trading Protocol (IOTP) [17], among other cases. Between other domains in which both formalisms have been applied together we may find the verification of web-services composition [19] [20] or vehicular traffic control systems [18], just to mention a few.

III. FORMAL DEFINITIONS

This section presents some formal definitions of the models and data structures that are manipulated through the functionalities of CPN Tools and Prosega/CPN. In particular, it is formulated how it can be derived an occurrence graph (*OG*) from a CPN model, and afterwards is explained how can it be generated a Finite-state Automaton (*FSA*) from a CPN's occurrence graph. The following formulations are based in the work done in [8]. Albeit CPNs are managed in this work; for the formal definition it has been rather convenient to generalize the type into a High-level Petri Net (i.e. for proving further theorems regarding the relationship between an *OG* and a *FSA* as described in [8]). Hence, we firstly take the definition of a High-level Petri net (*HLPN*) [21].

Definition 1. A High-level Petri net is a structure of the form $HLPN = (P, T, D; Type, Pre, Post, M_0)$ where

- P is a finite set of Places;
 - T is a finite set of Transitions, $P \cap T = \emptyset$;
 - D is a non-empty finite set of non-empty domains where each element of D is called a type;
 - $Type : P \cup T \rightarrow D$ is a function used to assign types to places and to determine transition modes;
 - $Pre, Post : TM \rightarrow \mu PLACE$ are the pre and post mappings with
 - $TM = \{(t, m) \mid t \in T, m \in Type(t)\}$, the set of transition modes;
 - $PLACE = \{(p, g) \mid p \in P, g \in Type(p)\}$, the set of elementary places.
 - $M_0 \in \mu PLACE$ is a multiset called the initial marking of the net.
- $\mu PLACE$ is the set of all possible multisets of $PLACE$.

For the analysis of a High-level Petri net it is generated an occurrence graph (*OG*). We consider that an *OG* can be defined as a labelled and rooted directed graph, where the nodes of the graph represent *markings* of the Petri Net, and the directed arcs represent the *transition modes* (or binding elements [2]) that can *occur* in all executions from the initial marking.

The root of the graph refers to a node which is considered as the initial state. In addition, the arcs of an *OG* may be labelled by the transition modes. Thus, we start by defining a labelled and rooted directed graph, and then we give the definition of an *OG* associated to a *HLPN*.

Definition 2. A labelled directed graph, with v_0 as the root node, is a triple $G = (V, L, E)$ where

- V is a finite set of vertices or nodes; $v_0 \in V$ represents the root or initial node.
- L is a set of labels;
- $E \subseteq V \times L \times V$ is a set of labelled directed edges.

Definition 3. An occurrence graph of a *HLPN* with an initial marking M_0 , is a labelled and rooted directed graph $OG = (V, TM, A)$ where

- V is the set of markings reachable from M_0 (the reachability set); $M_0 \in V$ represents the initial marking (root node).
- TM is the set of transition modes of the *HLPN*;
- $A = \{(M, tm, M') \in V \times TM \times V \mid M \xrightarrow{tm} M'\}$ is the set of arcs (directed edges) labelled by transition modes.

Remark. $M \xrightarrow{tm} M'$ indicates the occurrence of a transition mode $tm \in TM$ in a marking M which results in a new marking M' .

However, when we are only interested in the transition names, then the arcs of the *OG* are just labelled with such transitions names rather than the transition modes (binding elements). For example this is useful when it is just required to understand which user observable events (service primitives) may lead from a state of the system to another one; instead of transition modes which involve the parameters binded to such events.

In addition, when we are also interested in the identification of the markings for the nodes of the *OG*, rather than the marking details, we introduce an injection $I : [M_0] \rightarrow \mathbb{N}$. This function maps the set of reachable markings from M_0 (denoted as $[M_0]$) into the set of natural numbers.

Giving the described abstractions for transitions and markings, we consider the definition of an abstract *OG*.

Definition 4. An abstract *OG* of a *HLPN* with an initial marking M_0 , is a labelled and rooted directed graph $OG = (V, T, A)$ where

- $V = \{I(M) \mid M \in [M_0]\}$ is the set of nodes; $I(M_0) \in V$ represents the root or initial node.
- T is the set of transitions of the *HLPN*;
- $A = \{(I(M), t, I(M')) \in V \times T \times V \mid (t, m) \in TM, M \xrightarrow{(t, m)} M'\}$ is the set of arcs labelled by transition names.

We point out that the abstract occurrence graph *OG* defined above is *finite*. i.e. It has a finite number of states. Indeed this is an important fact when dealing with real scenarios. This means that the corresponding Petri Net must be a bounded

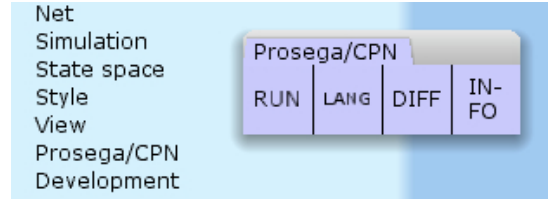


Figure 1. Tool palette of Prosega/CPN.

net [1], and hence a preliminary boundedness analysis on the Petri Net is performed.

Finally, it is presented a mapping from an abstract *OG* (Definition 4) into a Finite-state Automaton *FSA*.

We define a function $Prim : T \rightarrow SP \cup \{\epsilon\}$ that maps each transition of the *HLPN* to either an identifier name (i.e. an user observable event or service primitive name), or to an epsilon (i.e. an empty move); SP is the set of identifiers (for the user observable events or service primitive names) for the system that we are describing.

Definition 5. Given an abstract occurrence graph $OG = (V, T, A)$ it is derived the corresponding Finite-state Automaton $FSA = (V, SP, A_{SP}, v_0, F)$ where

- V is the set of nodes of the abstract *OG* (the states of the *FSA*);
- SP is the set of identifiers (for the user observable events or service primitive names) of the system of interest (the alphabet of *FSA*);
- $A_{SP} = \{(v, Prim(t), v') \mid (v, t, v') \in A\}$ is the set of transitions labelled by elements of SP or epsilons (the transition relation of the *FSA*);
- v_0 corresponds to the abstract initial marking (initial state of the *FSA*);
- $F \subseteq V$ is the set of final (acceptance) states.

Prosega/CPN performs the conversion of an *OG* as described in Definition 4 into a *FSA* as described in Definition 5. Moreover, this mapping between *OG* and the *FSA* allows the tool conveniently manage the generation of the language and the comparison between other *FSA*s.

IV. FUNCTIONALITIES

Prosega/CPN is an extension in CPN Tools. Thus, the user interacts with the application using a Graphical User Interface (GUI) through a tool palette added to CPN Tools (see Figure 1) - available under the Tool box entry [3]. The tool supports the generation of a minimized deterministic Finite-state Automaton (*FSA*) derived from the CPN's occurrence graph, the language generation, and the comparison between two different *FSA*s. We proceed to explain these functionalities in detail.

A. *FSA* generation

Once the occurrence graph (*OG*) from a CPN model is generated using the CPN Tools simulator [3], its associated Finite-state Automaton (*FSA*) can be generated and reduced

using the RUN tool (see Figure 1). To this aim, the following steps are performed: getting the transitions, and also dead markings of the *OG*, assigning identifiers to transitions (i.e. constructing the mapping *Prim* defined in Section III), reducing the *FSA*, and displaying the results. Here, we consider the structure of an abstract *OG* where the nodes are identified by numbers which represent the markings and the arcs are just labelled with the transitions rather than the binding elements (see Definition 4).

Firstly, the tool communicates with the CPN Tools simulator in order to obtain all the transitions and the dead markings (see Section V). The user interacts with the Prosega/CPN GUI to assign identifiers (corresponding to user observable events or service primitive names) to the model transitions (i.e. mapping elements from a set *SP*). The character 0 is considered as an epsilon (ϵ). Hence, any transition assigned with 0 is considered an epsilon transition (or empty move). Then, the user chooses the set of terminal states *F* for the *FSA* which may include nodes representing the dead markings or other nodes in the *OG*. Thereby, it is obtained a *FSA* in line with Definition 5.

For instance, Figure 2 shows the Prosega/CPN interface which supports the described operation. In particular, it is defined a *FSA* given a CPN's occurrence graph extracted from the use case in Section VI. The user assigns identifiers for the CPN transitions. For example, the identifier 1 to the transition *MACCrtConnReq*, which is in the CPN model page *CreatConnection*. Later, the user chooses the following nodes of the *OG* as terminal states: 1, 7, 8, 13, 26, 27, 31, 48 (some are not displayed in the figure due to window size limitation).

Afterwards, the modelled *FSA* is reduced by following the algorithm described in [22], which consists in performing the following operations over a *FSA*:

- removal of *epsilon* transitions (remove empties).
- removal of non-determinism (determinization).
- reduction by identifying and merging equivalent states (minimization).

Thereby, the algorithm produces as output a reduced deterministic *FSA* with a minimal number of states that is equivalent to the input automata. Finally, an interface showing the results of the *FSA* reduction is displayed to the user as shown in Figure 3. The interface shows general information about the reduced *FSA* (FSA Info), such as initial state and number of arcs, which may be relevant for the *FSA* analysis. It also includes a graphical representation of the *FSA* (FSA Image Preview), and the established mapping between the identification numbers/names assigned by the user and the transition names, which may be useful for debugging and verification of the model.

B. Language Generation

The language accepted by a *FSA* can be generated by using either the LANG tool in Figure 1 or the Generate Language button in Figure 3. The interface shown in Figure 4 is displayed to the user after it clicks on the LANG tool. Then the user can choose both the *FSA*, in plain text or in the compiled format [13], for which the language will

be generated and the corresponding symbol table file—for mapping the arc inscriptions with the symbols selected by the user. The language generator module generates the language *L* of the *FSA* by extension; if *L* is finite, the whole sequences are printed; otherwise a subset of the language, $L' \subseteq L$ is generated, as illustrated in Figure 5. In particular, L' is a set of symbol sequences whose symbols belong to different arcs in the *FSA*. Notice that some arcs of the *FSA* may be labelled with the same symbol. However, in the generation of each sequence, each arc of the *FSA* is visited just once.

Indeed, for generating each sequence accepted by the automaton it was developed an algorithm based on iterative Depth-first Search (DFS) which was implemented in the language generator component of Prosega/CPN (as mentioned in Section V). This component performs DFS between the initial state of the *FSA*, to each of the halt states. Hence, the symbols of the arcs visited through the path from the initial state to a specific halt state are printed, thereby representing a sequence accepted by the automaton.

In addition, this module supports a generator of random sequences of the language symbols, as shown in Figure 6, which may be useful when the language is infinite. For example, in Figures 5 and 6, we can see the following sequence of language symbols: 1,5 which corresponds to the sequence of actions (transitions): *MACCrtConnReq*, *MACCrtConnCf2* (as shown in the interface in Figure 4, where the user assigned an Id (language symbol) to each transition).

In particular, for generating each random sequence it is computed a random walk in the *FSA* from the initial state to any of the halt states. Whenever a halt state is visited, the walk will be terminated with a probability $\frac{p}{100}$ $0 < p \leq 100$, and the sequence of symbols which were collected throughout the visited path will be printed.

Thereby, in the Generate Random interface (Figure 6), the user can manipulate the average size of the randomly generated sequences of language symbols by entering the halt-rate parameter value *p*; therefore, if the value *p* is close to 0, the number of language symbols in each sequence may be big, while if *p* is close to 100, then the number of language symbols in each sequence may be small, thereby determining the length of each sequence. For instance, since the halt-rate parameter value in Figure 6 is 55, in that case the sizes of the sequences are medium.

C. FSA Difference

The user can use the DIFF tool to calculate the difference between two automata, F_A and F_B . This functionality, whose output interface is illustrated in Figure 7, generates a new automaton F_C which only accepts the sequence of symbols accepted by the first automaton F_A , and that are not accepted by the second one F_B . In particular, F_B must be an epsilon-free, deterministic finite automaton. This is useful to understand the sequences of languages symbols in which may differ two models; in this sense, as seen in Figure 7, this functionality allows to generating the language of F_C for getting such sequences in which may differ two models.

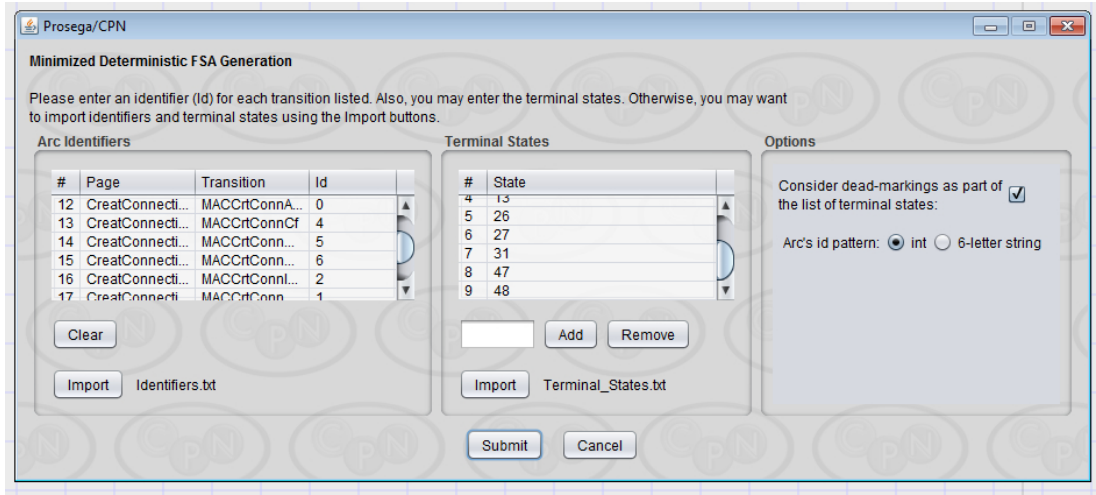


Figure 2. Initial Prosega/CPN interface used by the user for assigning Ids to transitions and entering terminal states.

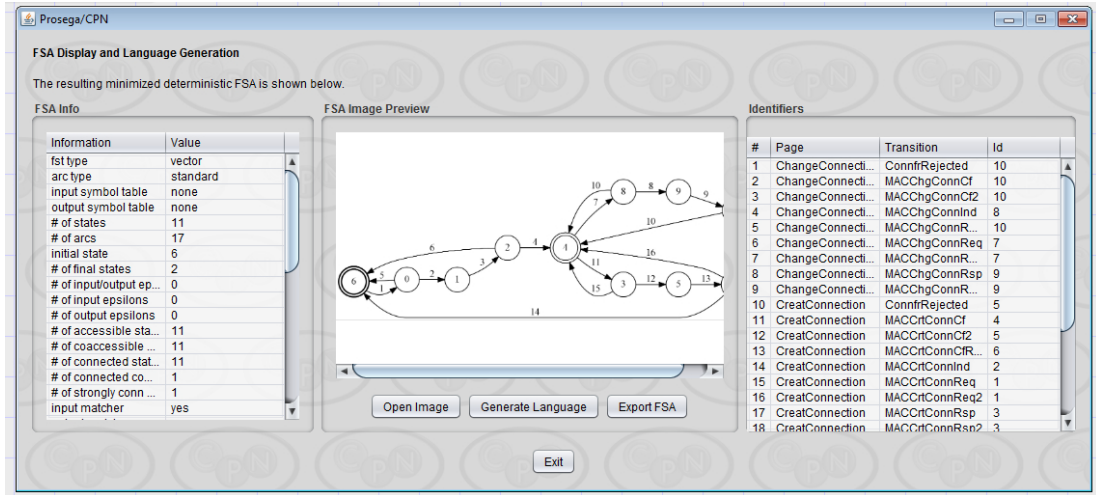


Figure 3. Interface showing the results of the *FSA* reduction process.

V. ARCHITECTURE

Prosega/CPN is implemented in Java programming language, so we use the new feature in CPN Tools 4 called Simulator Extensions [4] to add the software functionalities. Figure 8 shows the software architecture which illustrates the relationship among all the components of our tool, CPN Tools and the third-party components. Communication between the CPN Tools GUI and the simulator, and between the simulator and the Simulator Extensions is supported by the BIS (Boolean - Integer - String) protocol. Each protocol message is encoded using a number of booleans, integers, and strings as explained in [23]. In order to facilitate the development of Prosega/CPN we use some third-party libraries which implement many of the functions to manage and display the automata. In particular, we utilize OpenFST [13] [24] for *FSA* reduction and *FSA* difference, and Graphviz [25] for drawing the automata. On the other hand, we wrote the code for language generation (fsm2language) in C programming language [26].

The fsm2language implements the procedures for language generation and the computation of random sequences accepted by a *FSA*, that were described in Section IV. The bridge between the fsm2language component and the Prosega/CPN tool is supported by JNI (Java Native Interface) which enables a Java program to call native libraries written in C/C++ programming language.

VI. USE CASE

The IEEE 802.16 standard [27] is responsible for specifying and describing the air interface of Broadband Wireless Access Systems (BWA), and point-multipoint fixed/mobile wireless metropolitan area network. The standard is limited to the description of the Medium Access Control (MAC) and physical (PHY) layers. In overall, IEEE 802.16 provides great benefits for providing mass broadband wireless connectivity, allowing user mobility, mesh-mode network support, and even has been thought as an alternative for Internet-of-Things deployments. However, due to its inherent complexity, there are several

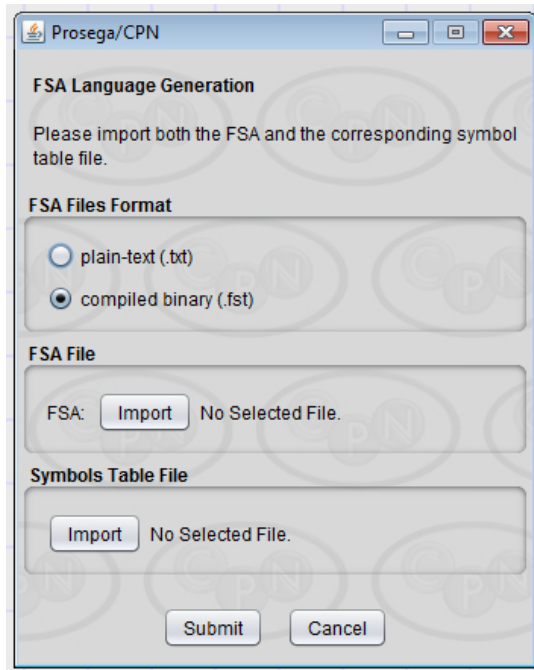


Figure 4. Language generation interface.

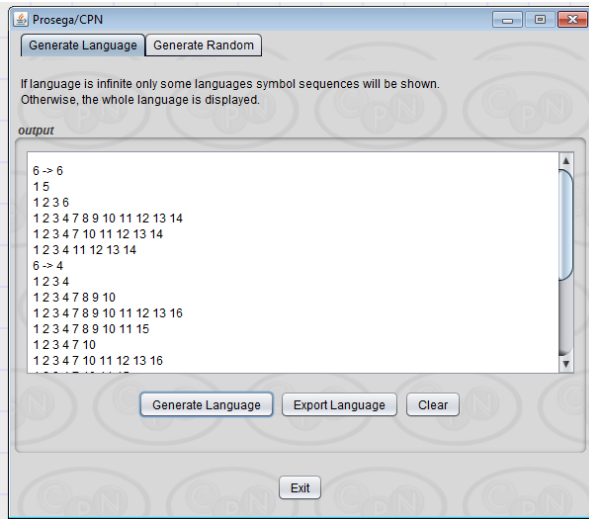


Figure 5. Interface showing part of the language accepted by the *FSA* generated in Figure 3.

parts of the specification that turn out to be ambiguous, difficult to understand and imprecise. In this context, Morales et al. [28] [29] has contributed establishing a formal model for a module of IEEE 802.16. In particular, it developed a formal verification of the MAC connection management service specification. To this aim, the Prosega/CPN tool has been used in conjunction with the Billington's protocol verification methodology [8]. Figure 9 illustrates the steps of the methodology; we proceed to explain such steps, and how they have been applied within our use case using CPN Tools and Prosega/CPN.

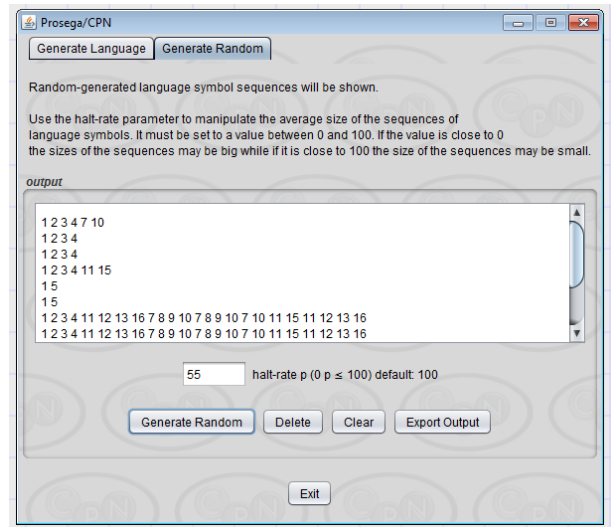


Figure 6. Interface showing some randomly generated sequences of language symbols.

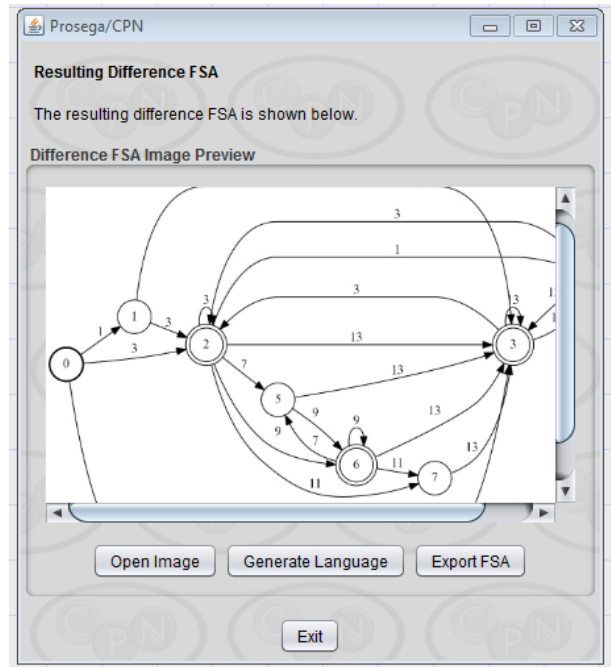


Figure 7. The interface shows the resulting difference *FSA* given two automata as parameters.

A. Service Definition

In Figure 9, the dashed box in the left represents the first step which consists in modelling the service specification of the system, and to define the services that it aims to provide (either to a higher layer or to another system entity). In the scenario of the IEEE 802.16 MAC layer, the service specification consists in a set of service primitives that the MAC sub-layer, responsible for connection management procedures, provides to the sub-layer on top of it. Each of these primitives correspond to one of the following procedures: The

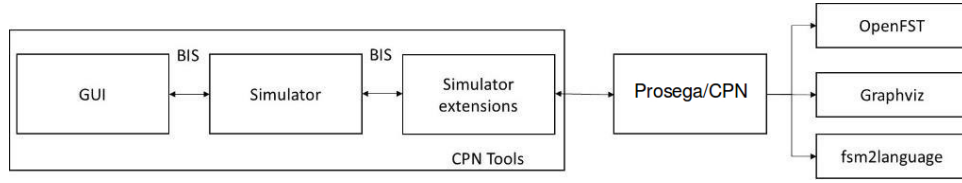


Figure 8. Prosega/CPN Architecture.

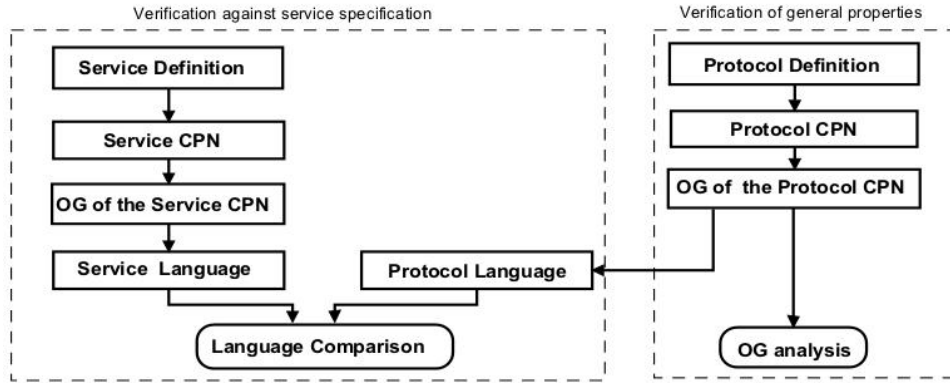


Figure 9. Steps within the protocol verification methodology proposed in [8].

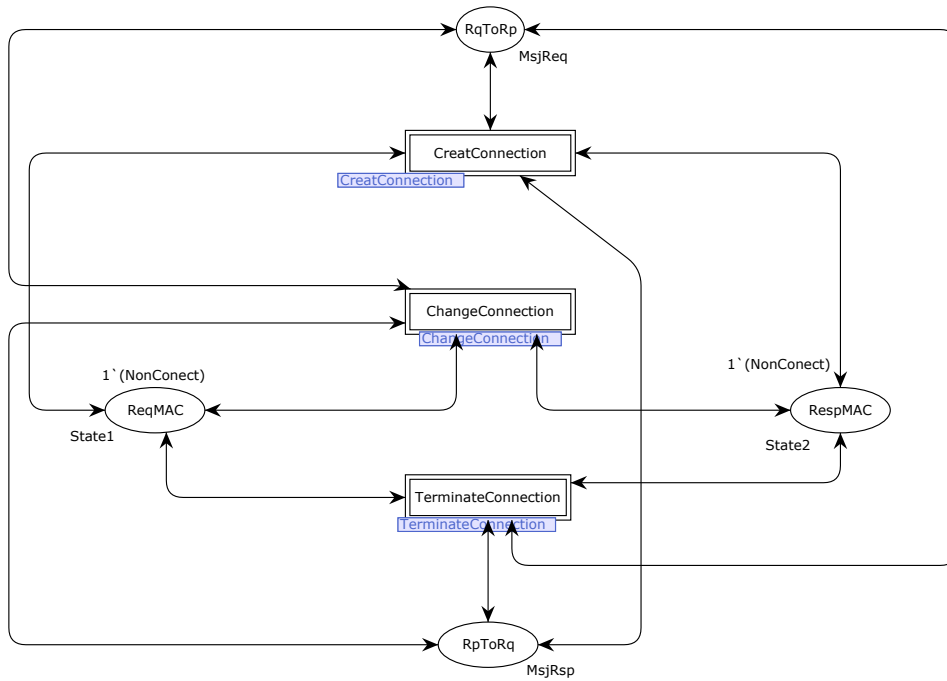


Figure 10. CPN model representing the hierarchical view for the processes of creation, changes and termination of connections between peer MAC entities in the IEEE 802.16 service specification.

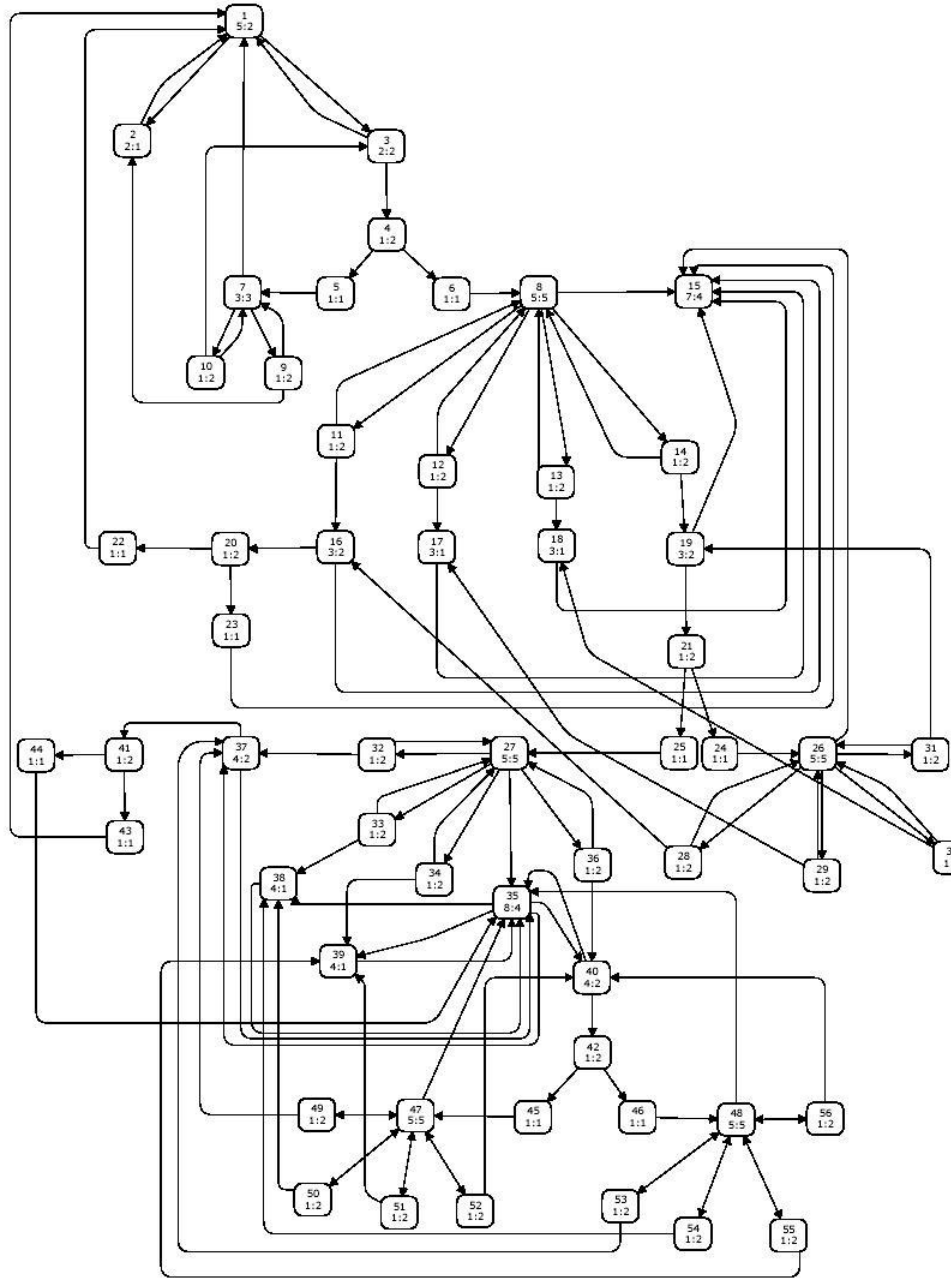


Figure 11. *OG* of the CPN model representing the IEEE 802.16 MAC connection management service specification.

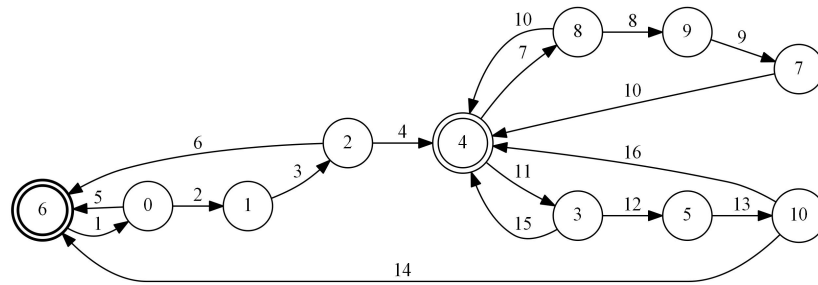


Figure 12. Minimized deterministic *FSA* generated from the *OG* illustrated in Figure 11.

establishment of a connection between communication peers, the connection maintenance (i.e. management of the dynamic network resources) and the termination of the connection by any of the communication peers.

B. Service CPN and OG

Using CPN Tools, it is created the CPN model of the service specification. Figure 10 presents the CPN main page which shows a top view of the model [2]. This top module is linked with the pages that model the service primitives that correspond to the establishment, maintenance, and termination of a connection through the transitions *CreatConnection*, *ChangeConnection*, and *TerminateConnection* respectively. Each of these pages of the model can be checked in [28]. Afterwards, it is generated the CPN's occurrence graph (*OG*), shown in Figure 11, which is the input for the *FSA* reduction feature of Prosega/CPN.

C. FSA Reduction

Once the service *OG* is generated, it is modelled as a *FSA* in line with Definitions 4 and 5. To this aim, it is used the RUN tool of Prosega/CPN for converting the *OG* into a *FSA* (as presented in Figure 2). For each transition of the CPN model, it is assigned a number value which represents the associated service primitive identifier (*Id*) (resembling the function *Prim* described in Section III). Transitions that are considered as empty moves (or internal events) are labelled with 0 (*epsilon* transitions). Later, there are assigned the terminal states. The assignation performed between all the model transitions and the service primitive identifiers as well as the decision of the terminal states can be fully checked in [28]. Afterwards, the *FSA* is minimized following the procedure explained in Section IV. Figure 12 presents the minimized deterministic *FSA* (exported from the output/analysis interface of the RUN tool previously presented in Figure 3).

D. Language Generation

The service language (the set of sequences of service primitives) is generated using Prosega/CPN as explained in Section IV —utilizing *FSA* minimization (RUN tool) and *FSA* language generation (LANG tool). Figure 5 presented some sequences that are accepted by the *FSA*. In addition, Table I shows the identifier selected for each primitive service [28]. For example, the sequence of language symbols 1, 2, 3, 4, 7, 8, 9, 10 represent the service primitives invoked by the protocol entity in top of the MAC for the successful establishment and maintenance (change of a communication resource) of the connection. In overall, the minimized *FSA* generated by Prosega/CPN provides a compact description of the possible sequences of service primitives, and allows to remove complexity from the model, which allows the language to present a clear specification of the service that the system provides.

Table I
SERVICE PRIMITIVES ON THE IEEE 802.16 MAC LAYER AND THEIR CORRESPONDING IDENTIFICATION NUMBER [28].

| Service Primitive | Id |
|---------------------------------------|------------|
| MAC_CREAT_CONNECTION.Request | 1 |
| MAC_CREAT_CONNECTION.Indication | 2 |
| MAC_CREAT_CONNECTION.Response | 3 |
| MAC_CREAT_CONNECTION.Confirmation | 4, 5, 6 |
| MAC_CHANGE_CONNECTION.Request | 7 |
| MAC_CHANGE_CONNECTION.Indication | 8 |
| MAC_CHANGE_CONNECTION.Response | 9 |
| MAC_CHANGE_CONNECTION.Confirmation | 10 |
| MAC_TERMINATE_CONNECTION.Request | 11 |
| MAC_TERMINATE_CONNECTION.Indication | 12 |
| MAC_TERMINATE_CONNECTION.Response | 13 |
| MAC_TERMINATE_CONNECTION.Confirmation | 14, 15, 16 |

E. Further Steps

The second part of the methodology (dashed box in the right of Figure 9) concerns to the modelling of the protocol, and its comparison against the service specification through language equivalence. These further steps are still in progress within the research work [28]. The modelling of the protocol consists in constructing the CPN model which describes the protocol procedures which are performed when a service primitive is invoked by a higher entity of the system. Later, it is generated the *OG* associated to this CPN model. On the one hand, behavioral properties of the protocol may be analyzed through the *OG*. On the other hand, the *OG* may be reduced into a minimized deterministic *FSA*. i.e. using again the RUN tool of Prosega/CPN. Then, the *FSA* of the service specification may be compared with the *FSA* of the protocol. i.e. using the DIFF function of Prosega/CPN - see Figure 7. Finally, the language of the difference *FSA* may be generated in order to determine language equivalence between the service and the protocol. Thus, we can determine the sequences of service primitives which are in the protocol specification but are not in the service specification. It is important to know if the service specification meets the protocol specification, since it is not desirable to have a service requirement from the service user which cannot be met by the protocol. In addition, it may not be wanted a service provided by the protocol which actually it is never required by the user.

VII. CONCLUSIONS

This work has presented Prosega/CPN. The tool is an extension of CPN Tools for supporting several operations for *FSA*-based analysis and system verification. The tool provides a feature for generating a minimized deterministic Finite-state Automaton (*FSA*) from a CPN's occurrence graph (*OG*). It includes as well operations for language generation, and for automata comparison. These functionalities are supported

taking advantage of consolidated third-party components such as OpenFST and Graphviz. In addition, we developed a module for language generation.

Prosega/CPN has been integrated within the CPN Tools GUI using the Simulation Extensions (new feature in the last version of CPN Tools) component whose development has been driven by the demand of many research works to suitably integrate Coloured Petri Nets with other formalisms [4]. In particular, the integration between CPNs and *FSA* was not existing within CPN Tools, and the application of this multi-formalism strategy has shown its merits in many published papers, specially from the domain of protocol verification. Furthermore, other works may be benefited from this *FSA*-based verification; for example, as presented in our use case, the analysis of an equivalent reduced *FSA* provides a compact and clear description of the possible user observable events (service primitive calls) rather than to deal with the analysis of the *OG*, thereby allowing to reduce the time complexity when it may be required to check the behavioral properties of the system through the *FSA*.

As future work, the tool will keep providing support within the further steps of the formal verification work of the IEEE 802.16 standard, regarding to the MAC connection management procedures. On the other hand, as another further direction for the tool enhancement, the tool has been thought to be tested in other domains; indeed, as it has been stated, Prosega/CPN can be used in other cases where *FSA* may be required, and within the verification of other systems whose analysis may involve the comparison of models at different levels of abstraction. This future work on other use cases will be able to keep maturing the tool. i.e. integrating new operations/features for automata manipulation, and testing the tool performance in terms of scalability, among other key facts. In addition, it has been considered to keep exploiting more capabilities offered by the Simulator Extensions channel; for example, to be able draw and manually edit a *FSA* in the CPN Tools canvas, instead of only using the Graphviz support for automata drawing.

REFERENCES

- [1] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, April 1989.
- [2] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Berlin, Heidelberg: Springer-Verlag, 2009.
- [3] "CPN Tools - A tool for editing, simulating, and analyzing Coloured Petri Nets," <http://www.cpn-tools.org/>.
- [4] M. Westergaard, "CPN Tools 4: Multi-formalism and Extensibility," in *Application and Theory of Petri Nets and Concurrency*. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 400–409.
- [5] M. E. Villapol, "Modelling and Analysis of the Resource Reservation Protocol Using Coloured Petri Nets," Ph.D. dissertation, University of South Australia, Australia, December 2003.
- [6] S. Gordon, L. M. Kristensen, and J. Billington, "Verification of a Revised WAP Wireless Transaction Protocol," in *Application and Theory of Petri Nets and Concurrency*. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 182–202.
- [7] B. Han, "Formal Specification of the TCP Service and Verification of TCP Connection Management," Ph.D. dissertation, University of South Australia, Australia, April 2004.
- [8] J. Billington, G. E. Gallasch, and B. Han, *A Coloured Petri Net Approach to Protocol Verification*. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 210–290.
- [9] G. Gallasch and L. M. Kristensen, "Comms/CPN: A Communication Infrastructure for External Communication with Design/CPN," January 2001.
- [10] M. Westergaard and K. B. Lassen, "The BRITNeY Suite Animation Tool," in *Applications and Theory of Petri Nets and Concurrency*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 431–440.
- [11] M. Westergaard, "Access/CPN 2.0: A High-Level Interface to Coloured Petri Net Models," in *Application and Theory of Petri Nets and Concurrency*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 328–337.
- [12] AT&T Researchers - Inventing the Science Behind the Service, <http://www.research.att.com/evergreen/portfolio/>.
- [13] "OpenFst Library," <http://www.cpn-tools.org/>.
- [14] M. Hulden, "Foma: A Finite-state Compiler and Library," in *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics: Demonstrations Session*. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 29–32.
- [15] A. Almeida, M. Almeida, J. Alves, N. Moreira, and R. Reis, "FAdo and GUITar: Tools for Automata Manipulation and Visualization," in *Implementation and Application of Automata*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 65–74.
- [16] S. H. Rodger, *JFLAP: An Interactive Formal Languages and Automata Package*. USA: Jones and Bartlett Publishers, Inc., 2006.
- [17] C. Ouyang and J. Billington, "Formal Analysis of the Internet Open Trading Protocol," in *Applying Formal Methods: Testing, Performance, and M/E-Commerce*. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 1–15.
- [18] S. Barzegar, M. Davoudpour, M. R. Meybodi, A. Sadeghian, and M. Tirandazian, "Traffic Signal Control with Adaptive Fuzzy Coloured Petri Net Based on Learning Automata," in *Annual Meeting of the North American Fuzzy Information Processing Society*, July 2010, pp. 1–8.
- [19] N. Danapaluk, E. Ilavarasan, N. Kumar, and S. K. Dwivedi, "Ratification strategy for web service composition using CPN: A survey," in *IEEE International Conference on Computational Intelligence and Computing Research*, December 2013, pp. 1–4.
- [20] J. Zhu, K. Zhang, and G. Zhang, "Verifying Web Services Composition based on LTL and colored Petri Net," in *6th International Conference on Computer Science Education*, August 2011, pp. 1127–1130.
- [21] ISO/IEC, "High-level Petri Nets - Part 1: Concepts, Definitions and Graphical Notation," Software and Systems Engineering, ISO/IEC FDIS 15909-1. Final Draft International.
- [22] W. A. Barrett and J. D. Couch, *Compiler Construction: Theory and Practice*. Chicago, Illinois: Science Research Associates Inc., 1979.
- [23] M. Westergaard, "CPN Tools 4 Extensions: Part 4: Advanced Communication and Debugging," <https://westergaard.eu/2013/11/cpn-tools-4-extensions-part-4-advanced-communication-and-debugging/>, November 2013, Blog entry.
- [24] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri, "OpenFst: A General and Efficient Weighted Finite-State Transducer Library," in *Implementation and Application of Automata*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 11–23.
- [25] "Graphviz - Graph Visualization Software," <http://www.graphviz.org/>.
- [26] J. C. Carrasquel, "Java/PROSEGA: An extension in CPN Tools for generating languages accepted by FSA and minimized deterministic FSA from a state space," Central University of Venezuela, Caracas, Venezuela, Tech. Rep., October 2015.
- [27] IEEE 802.16 Working Group on Broadband Wireless Access Standards, "IEEE Std. 802.16e-2005. Local and Metropolitan Area Network. Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems."
- [28] A. V. Morales and M. E. Villapol, "Towards Formal Specification of the Service in the IEEE 802.16 MAC Layer for Connection Management," in *Proceedings of the 9th WSEAS International Conference on Computational Intelligence, Man-machine Systems and Cybernetics*. World Scientific and Engineering Academy and Society (WSEAS), 2010, pp. 140–146, Mérida, Venezuela.
- [29] A. V. Morales and M. E. Villapol, "Reviewing the Service Specification of the IEEE 802.16 MAC Layer Connection Management: A Formal Approach," in *CLEI Electronic Journal*, vol. 16, August 2013, pp. 1–12.

Simulating Behavior of Multi-Agent Systems with Acyclic Interactions of Agents

Roman A. Nesterov^{*†}, Alexey A. Mitsyuk^{*}, Irina A. Lomazova^{*}

^{*}National Research University Higher School of Economics, 20 Myasnitskaya Ulitsa, 101000 Moscow, Russia

[†]Univeristà degli Studi di Milano-Bicocca, 1 Piazza dell'Ateneo Nuovo, 20126 Milan, Italy

E-mail: {rnesterov, amitsyuk, ilomazova}@hse.ru

Abstract—In this paper, we present an approach to model and simulate models of multi-agent systems (MAS) using Petri nets. A MAS is modeled as a set of workflow nets. The agent-to-agent interactions are described by means of an interface. It is a logical formula over interaction rules specifying the order of inner agent actions. Our study considers positive and negative interaction rules. In this work, we study only interfaces which describe acyclic agent interactions. We propose an algorithm for simulating the MAS with respect to a given interface. The algorithm is implemented as a ProM 6 plug-in that allows one to generate a set of event logs. We suggest our approach to be used for evaluating process discovery techniques against the quality of obtained models since this research area is on the rise. The proposed approach can be used for process discovery algorithms concerning internal agent interactions of the MAS.

Index Terms—Petri nets, multi-agent systems, interaction, interfaces, simulation, event logs

I. INTRODUCTION

Process discovery has been actively developed over recent years [1]. Many algorithms for the automatic model synthesis from event logs have been proposed [2]–[7]. They produce process models in different notations. These can be Petri nets [3], [6], [7], fuzzy models [2], heuristics nets [4] or BPMN models [5] and many others (see [8] for the comprehensive review of process discovery algorithms).

Discovering process models from event logs helps to use information about users and system runtime behavior for proper specification, design, and maintenance of software systems [9], [10]. This topic is increasingly attracting the attention of researchers [11]–[14]. In particular, application of process mining techniques to distributed and multi-agent software systems [15], [16] is interesting and important.

The main drawback of most algorithms is that they are not appropriate for modeling highly concurrent systems. In particular, these are *multi-agent systems* (MAS). Such a system consists of multiple agents executing their work independently and interacting via predefined interfaces. It makes sense to use compositional approaches to model MASs. Fortunately, such approaches have been proposed within recent years [17], [18].

The overwhelming majority of process discovery algorithms employ different heuristics. That is why, testing is used to evaluate their efficiency and validity [8]. It is performed using real-life and artificially generated event logs with suitable characteristics. The latter are prepared using *event log generators*.

In this paper, we describe a new event log generator that aims at preparing artificial event logs for MASs. We model

individual agents using workflow nets, whereas interfaces are specified using special formulae. They are constructed using a declarative formalism that we introduce to describe basic *asynchronous* interactions between agents. Based on agent models and a declarative interface formula our generator derives the operational semantics that describes a MAS behavior. We show that both representations of a MAS are equivalent, i.e. they have the same set of possible model runs. Thus, this semantics can be used to simulate the model and generate event logs.

The **main contributions of this paper** are:

- 1) a formalism for a declarative description of the requirements for agent interactions is defined;
- 2) the operational semantics representing the behavior of a multi-agent system with declarative requirements for interactions of agents is defined;
- 3) an algorithm for generating event logs from given agent models and declarative constraints on their interactions based on the operational semantics is developed;
- 4) the approach is implemented as a prototype software and evaluated.

This paper is structured as follows. The next section gives an overview of existing approaches for generating event logs and simulating process models. Section III introduces main notions used in the paper. In Section IV, we describe our approach to modeling multi-agent systems with the help of Petri nets. Implementation details are discussed in Section V, and Section VI concludes the paper.

II. RELATED WORK

Process Logs Generator PLG2 [19] is one of the most popular tools for generating well-structured process models represented by dependency graphs. The tool constructs models using randomly generated context-free grammars. The user should specify desired characteristics of models: a size, a number of choices, hierarchy blocks etc. Afterwards, the obtained model can be used to generate an event log.

Another tool that aims at randomized event log generation is *PT and Log Generator* [20]. It generates random process trees (well-structured models), which contain desired number of specified workflow patterns. In particular, generated models can be constructed from sequences, AND-/XOR-/OR- splits and joins, structured loops. The algorithm can also randomly insert elements representing activities. The tool also generates

the desired number of event logs from automatically constructed models.

The problem of the randomized process model generation has been also considered by Yan, Dijkman, and Grefen in [21]. Their approach does not consider event log generation.

The main goal of the tools discussed above is the randomized testing using sets of models and event logs. However, in some cases there is a need to generate event logs from specific process models that have been prepared on the basis of the real data or expert knowledge. If this is the case, one can use the tool *GENA* [22]. It aims at generating sets of event logs from a Petri net model. The approach allows users to use preferences to influence a control-flow and to artificially introduce a randomized noise into an event log. The improved version of *GENA* can generate event logs from BPMN 2.0 models [23]. Most basic BPMN constructs are supported: tasks, gateways, messages, pools, lanes, data objects.

Colored Petri nets can also be used to generate event logs [24]. Authors have developed the extension for *CPN Tools* that can generate randomized event logs based on a given colored Petri net. The main drawback of this approach is that it implies writing *Standard ML* scripts, which leads to possible problems during tool adaptation for a specific task. Moreover, this approach and *GENA* do not support multi-agent systems with independent asynchronous agents.

Declarative process models can also be used to generate event logs [25]. This approach is based on construction of a finite automaton using a *Declare* process model. The tool can generate a specified number of strings accepted by this automaton. Strings are generated using the automaton and its randomized execution. Afterwards, each string is transformed into a log trace with necessary attributes. This tool is useful, when the only information about the process is the set of constraints. This approach is also not appropriate for the MAS simulation as we suggest, because it does not support the imperative control-flow description of individual agents.

In this paper, we propose an extension to the *GENA* tool that can be used for generating event logs by simulating MAS models, because the tools described above cannot fully support this feature.

III. PRELIMINARIES

Let \mathbb{N} denote the set of all non-negative integers, A^+ — the set of all finite non-empty sequences over a set A , and $A^* = A^+ \cup \{\epsilon\}$, where ϵ is the empty sequence. For a subset $B \subset A$ the projection of $\sigma \in A^*$ on set B , denoted $\sigma|_B$, is the subsequence of σ including all elements belonging to B .

A. Petri Nets

A **Petri net** is a triple $N = (P, T, F)$, where P and T are two disjoint sets of places and transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation. Pictorially, places are shown by circles, transitions — by boxes, whereas the flow relation is depicted using directed arcs (see Fig. 1 for an example).

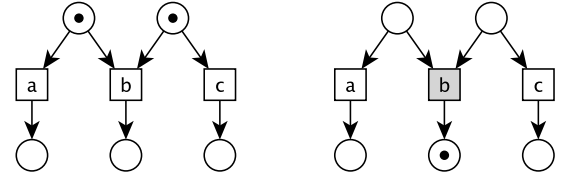
We suppose that transitions of a Petri net are labeled with activity names from $\mathcal{A} \cup \{\tau\}$ where \mathcal{A} is a set of visible activity

names, and τ is a label for an invisible action. Labels are assigned to transitions via a labeling function $\lambda: T \rightarrow \mathcal{A} \cup \{\tau\}$.

A marking (*state*) of a Petri net N is a function $m: P \rightarrow \mathbb{N}$ assigning numbers to places. A marking m is designated by putting $m(p)$ black dots into each place p . By m_0 we denote the initial marking.

Let $X = P \cup T$. For $x \in X$, $\bullet x = \{y \in X \mid (y, x) \in F\}$ is the set of *input* nodes of x in N , and $x^\bullet = \{y \in Y \mid (x, y) \in F\}$ is the set of its *output* nodes.

A marking m **enables** a transition $t \in T$ iff there is at least one token in all input for t places. An enabled transition may **fire** yielding a new marking m' (denoted $m[t]m'$), consuming one token from each of its input places and producing a token into each of its output places (Fig. 1b).



(a) initial marking (b) transition b fires

Fig. 1. A Petri net

A sequence $w = t_1 t_2 \dots t_n$ over T is a **firing sequence** iff $m_0[t_1]m_1[t_2] \dots m_{n-1}[t_n]m_n$ (denoted $m_0[w]m_n$).

Let $w = t_1 t_2 \dots t_n$ be a firing sequence in the net N , λ — a labeling function over a set of activity names \mathcal{A} . Define $\lambda(w) = \lambda(t_1)\lambda(t_2) \dots \lambda(t_n)$. Then $\lambda(w)|_{\mathcal{A}}$ is called an (*observable*) **run** in N .

A marking m is **reachable** iff $\exists w \in T^*$, s.t. $m_0[w]m$. A reachable marking is called **dead** if it does not enable any transition.

Workflow nets (WF-nets) form a subclass of Petri nets used for business process modeling.

A Petri net $N = (P, T, F, m_0)$ is a WF-net iff:

- 1) there is a single source place i and a single sink place f , s.t. $\bullet i = f^\bullet = \emptyset$;
- 2) each node in $P \cup T$ lies on a path from i to f .

The initial marking m_0 for a WF-net contains exactly one token in its source place i .

B. Event Logs

A **multiset** over a set A is a map $B: A \rightarrow \mathbb{N}$. The set of all multisets over A is denoted by $\mathcal{B}(A)$.

Let \mathcal{A} be a set of activity names. A **trace** σ over \mathcal{A} is defined as a finite non-empty sequence over \mathcal{A} . An **event log** L over \mathcal{A} is a finite multiset of traces, i.e. $L \in \mathcal{B}(\mathcal{A}^+)$.

IV. MODELING MULTI-AGENT SYSTEMS

In this section, we present a formalism for modeling multi-agent systems consisting of several asynchronously interacting agents.

A model for a system of k agents will consist of k WF-nets N_1, N_2, \dots, N_k , representing behavior of individual agents

(called **agent nets**), and constraints on their asynchronous interaction \mathcal{I} (called **interface**). We assume that transitions of agent nets have individual labels. In other words, different agents implement different activities. We also assume that agent interactions are **acyclic**, namely, activities in interaction constraints do not belong to cycles and therefore occur in each system run not more than once.

Interfaces are defined as positive logical formulae over atomic constraints. Let us give the exact definitions.

Let N_1, N_2, \dots, N_k be agent nets with pairwise disjoint sets of activity names $\lambda_1(T_1), \lambda_2(T_2), \dots, \lambda_k(T_k)$ respectively. We define two types of **atomic constraints**, namely, $A \triangleleft B$, and $A \triangleright B$, where A and B are activity names from two different sets, i.e. $A \in \lambda_i(T_i)$, $B \in \lambda_j(T_j)$, and $i \neq j$.

The **validity** of atomic constraints for a given trace σ over the set of activity names $\mathcal{A} = \lambda_1(T_1) \cup \lambda_2(T_2) \cup \dots \cup \lambda_k(T_k)$ is defined as follows:

$\sigma \models A \triangleleft B \Leftrightarrow$ if B occurs in σ , then A occurs before B ;
 $\sigma \models A \triangleright B \Leftrightarrow A$ does not occur before B .

When $\sigma \models \phi$, we say that ϕ is **valid** for σ , and σ **satisfies** ϕ .

The validity of the atomic constraints has a natural interpretation. The constraint $A \triangleleft B$ means that B should be always preceded by A , e.g. a message can be received only if it has already been sent. Thus, $A \triangleleft B$ is valid for a trace $\sigma = \dots A \dots B \dots$ and is not valid for a trace $\sigma' = \underbrace{\dots}_{\text{except } A} B \dots$.

The constraint $A \triangleright B$ means that B cannot occur, if A has happened before, e.g. if a message was already sent by mail, we should not fax it. A trace $\sigma' = \underbrace{\dots}_{\text{except } A} B \dots$ satisfies this constraint, and a trace $\sigma = \dots A \dots B \dots$ does not satisfy it. Note, however, that atomic constraints are not negations of each other. Both $A \triangleleft B$ and $A \triangleright B$ are valid for a trace which does not contain B .

Now a language of interface constraints is defined by the following grammar rules:

$$\begin{aligned} \text{Atom} &::= A \triangleleft B \mid A \triangleright B \\ \phi &::= \text{Atom} \mid \phi \vee \phi \mid \phi \wedge \phi, \end{aligned}$$

where Atom is an atomic constraint, and ϕ — a constraint formula.

Validity of a constraint formula for a given trace is defined in a standard way:

$$\begin{aligned} \sigma \models \phi_1 \wedge \phi_2 &\Leftrightarrow \sigma \models \phi_1 \text{ and } \sigma \models \phi_2 \\ \sigma \models \phi_1 \vee \phi_2 &\Leftrightarrow \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \end{aligned}$$

Let L be an event log over a set \mathcal{A} of activity names, ϕ — a constraint formula, then ϕ is **valid** for L iff ϕ is valid for each trace in L .

Interface formulae allow us to express different useful interaction constraints, e.g. the formula $\phi = (A \triangleright B \wedge B \triangleright A)$ describes a conflict between A and B , i.e. A and B cannot occur in the same trace.

Recall that a MAS model consists of k agent nets N_1, N_2, \dots, N_k , where $N_i = (P_i, T_i, F_i, m_0^i, \lambda_i)$, and a constraint formula \mathcal{I} (interface) with atomic constraints that defines the relations on activities of different agents.

It is easy to see that the **union** of Petri nets (considering several disjoint graphs as one disconnected graph) is also a Petri net. So, we can consider k agent nets as a single Petri net N . Recall that a run for a Petri net N is a sequence of activity names, corresponding to a firing sequence of N , and a trace from the related event log. Then, for a MAS model $S = (N_1, N_2, \dots, N_k, \mathcal{I})$ a **run** is defined as a run ρ in N , satisfying \mathcal{I} , i.e. $\rho \models \mathcal{I}$.

The following proposition is the immediate consequence of the definitions.

Proposition 1: Let $S = (N_1, N_2, \dots, N_k, \mathcal{I})$ be a MAS model, and ρ — a run in S . Then for all i the projection $\rho|_{N_i}$ of the run ρ on transitions of an agent net N_i is a run in N_i .

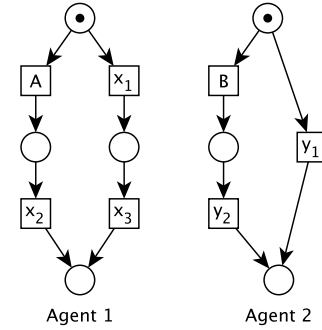


Fig. 2. A MAS with two interacting agents

Consider as an example the system in Fig. 2 with $\mathcal{I} = (A \triangleright B) \wedge (B \triangleright A)$, which means that B conflicts with A . Consider a run $\sigma = x_1 B y_2 x_3$ satisfying \mathcal{I} . Projecting σ on agent nets gives traces $x_1 x_3$ and $B y_2$ which are runs of agent nets. This property will be used for designing the simulation algorithm presented in the next section.

V. SIMULATING MAS PROCESS MODELS

In this section, we describe an algorithm for simulating MAS models. This algorithm was implemented as a ProM 6 plug-in, developed on the basis of GENA tool [22].

A. An Interface-Driven Firing Rule

A constraint formula in a MAS model defines declarative restrictions on the model's behavior. To simulate the model behavior, we need to define operational semantics for MAS models based on a special firing rule for selecting and executing the next step in the run of the model. We call this rule an **interface-driven firing rule** to distinguish it from the standard Petri net firing rule. Naturally, this rule should be consistent with the declarative definitions of MAS models behavior.

Let $S = (N, \mathcal{I})$ be a MAS model, where a Petri net $N = (P, T, F, m_0, \lambda)$ is a union of all agent nets.

Firstly, we convert \mathcal{I} to a disjunctive normal form (DNF) using standard logical laws. Then, an interface $\mathcal{I} = \bigvee_{j=1}^n C_j$,

where $C_j = \bigwedge_{\ell=1}^m S_\ell$, and S_ℓ is an atomic constraint. By abuse of notation, we denote by \mathcal{I} also the set of its conjuncts, and by C_j — the set of atomic constraints in a conjunct C_j .

Obviously, a trace σ satisfies \mathcal{I} iff $\exists C_j \in \mathcal{I}: \sigma \models C_j$, i.e. σ satisfies at least one conjunct in \mathcal{I} . So, to generate a model run, we choose a conjunct C_j and fire transitions of N only if they do not violate C_j .

Then we define $T_{\mathcal{I}} \subseteq T$ to be the set of transitions involved in agent interaction, i.e. $t \in T_{\mathcal{I}}$ iff $\lambda(t)$ occurs in \mathcal{I} . We call transitions from $T_{\mathcal{I}}$ **interface transitions**. Independent transitions from $T \setminus T_{\mathcal{I}}$ fire according to the standard firing rule for Petri nets. The firing of interface transitions is restricted by the constraint formula. To check whether firing of a transition t violates C_j , we keep the current historical run, i.e. a sequence of already fired activities. When a transition $t \in T_{\mathcal{I}}$ is enabled according to the standard Petri net firing rule at a current marking m , and an atomic constraint $A \triangleleft \lambda(t)$ occurs in C_j , then t is defined to be enabled only if A occurs in the current run. Similarly, if $A \triangleright \lambda(t)$ occurs in C_j , then t is enabled only if A does not occur in the current run. Otherwise, a transition t is enabled in the model, when it is enabled in N .

Now the **operational semantics** of a MAS model $S = (N, \mathcal{I})$, where $N = (P, T, F, m_0, \lambda)$, and $\mathcal{I} = \bigvee_{j=1}^n C_j$, is defined by the following execution procedure.

Step 1. Choose nondeterministically a conjunct C in \mathcal{I} .

Step 2. Start with the initial marking m_0 and the empty sequence ϵ for a current run σ .

Step 3. For a current marking m and a current run σ repeat while there are enabled transitions of N :

- 1) compute the set T_{ok} of all transitions enabled at m , not violating constraints from C w.r.t. σ ;
- 2) choose nondeterministically a transition t from T_{ok} ;
- 3) fire t by changing the current marking to m' , $m[t]m'$, and adding $\lambda(t)$ to the run σ .

B. Event Log Generation

In this section we present an algorithm for generating an event log by simulating a MAS model.

Let $S = (N, \mathcal{I})$ be a MAS model, where $N = (P, T, F, m_0, \lambda)$ is a Petri net, and \mathcal{I} is in DNF. Firstly, for each conjunct C occurring in \mathcal{I} , we simulate S to check if it is possible to obtain a run σ satisfying C . If we cannot obtain such a run by simulating S , we exclude this conjunct. As a result, we come to a set of conjuncts $\mathcal{I}' \subseteq \mathcal{I}$ which can actually be satisfied by runs of S or an empty set if \mathcal{I} cannot be satisfied by runs of S . If $\mathcal{I}' = \emptyset$, then the simulation is terminated producing the empty event log L .

That is why, we can simulate S with respect to conjuncts occurring in \mathcal{I}' only. Starting a new *simulation iteration*, we randomly choose a conjunct from \mathcal{I}' and execute N according to the interface-driven firing rule.

The end user also specifies the final marking m_f which is actually a set of output places of agent nets. Apart from that, log generation is also regulated by the number of logs and traces in a log and by the maximum number of steps which can be done while generating a single trace (*maxSteps*).

Algorithm 1 is used for generating a single trace, which satisfies a nondeterministically chosen conjunct C from \mathcal{I}' .

Algorithm 2 is used for finding enabled transitions which do not violate constraints of C . Firstly, we find a set of transitions enabled at a reachable marking m according to the standard firing rule. Secondly, if m enables interface transitions, we check whether the current run $\sigma = \lambda(w)|_A$, s.t. $m_0[w]m$, satisfies constraints of C using the interface-driven firing rule. A run σ is a trace to be recorded into an event log L .

Algorithm 1: Single trace generation

Input: $N = (P, T, F, m_0, \lambda)$, \mathcal{I}' , m_f

Output: Trace σ , s.t. $\sigma \models \mathcal{I}'$

```

 $\sigma \leftarrow \epsilon$                                 /* current run */
 $m \leftarrow m_0$                             /* current marking */
 $i \leftarrow 1$                               /* step number */
 $C \leftarrow \text{pickRandomConjunct}(\mathcal{I}')$  /* chosen conjunct */
while ( $i \leq \text{maxSteps}$ )  $\wedge$  ( $m \neq m_f$ ) do
     $T_{ok} \leftarrow \text{findEnabledTransitions}(N, m, C, \sigma)$ 
    if  $T_{ok} \neq \emptyset$  then
         $t \leftarrow \text{pickRandomTransition}(T_{ok})$ 
         $m \leftarrow \text{fireTransition}(N, m, t)$ 
        if  $\lambda(t) \neq \tau$  then
            /* visible actions are recorded in  $\sigma$  */
             $\sigma \leftarrow \sigma + \lambda(t)$ ;  $i \leftarrow i + 1$ ;
        end
    else
         $\sigma \leftarrow \epsilon$ ; break;                /* dead marking */
    end
end

```

Algorithm 2: Function findEnabledTransitions

Input: $N = (P, T, F, m_0, \lambda)$, $m \in [m_0]$, $C \in \mathcal{I}'$, σ

Output: A set of transitions T_{ok} enabled w.r.t. C

```

 $T_m \leftarrow \text{stEnabledTransitions}(N, m)$  /* firing rule */
 $T_{ok} \leftarrow T_m \setminus T_{\mathcal{I}}$  /* non-interface transitions */
foreach  $t \in T_m \cap T_{\mathcal{I}}$  do /* check whether  $\sigma \models C$  */
    foreach  $S \in C$  do /* if right of  $S$  is  $\lambda(t)$  */
        if  $S = X \triangleleft \lambda(t)$  then
            if  $\sigma = uXv$  then /*  $u, v$  can be empty */
                 $T_{ok} \leftarrow T_{ok} \cup t$ ;
            end
        else if  $S = X \triangleright \lambda(t)$  then
            if  $\sigma \neq uXv$  then  $T_{ok} \leftarrow T_{ok} \cup t$ ;
            end
        end
    end
end

```

We do not show here how the transition firing is implemented. It is discussed in detail in [22] where the original GENA plug-in is described.

Consider an example based on the system shown in Fig. 2. Assume $\mathcal{I} = (A \triangleleft B) \vee (y_1 \triangleleft x_1 \wedge x_2 \triangleright y_1)$. $C = y_1 \triangleleft x_1 \wedge x_2 \triangleright y_1$ is chosen. We are at the initial marking, so the run is empty, i.e. $\sigma = \epsilon$. Enabled transitions are $\{A, x_1, B, y_1\}$.

However, x_1 cannot fire, since it should wait until y_1 is executed. Then, nondeterministically B fires. Subsequently, the run is $\sigma = B$, and enabled transitions are $\{A, x_1, y_2\}$, but x_1 still cannot fire. We can choose A to fire. Then, the trace is $\sigma = BA$, and the enabled transitions are $\{x_2, y_2\}$, which are not influenced by C . As a result, we can obtain a trace $\sigma = BAy_2x_2$ satisfying C , and the projections of σ on agent transitions, Ax_2 and By_2 , are the runs of agent nets.

C. Experimental Simulation

We have developed the extension for the ProM¹ plug-in GENA which implements the proposed simulation algorithm and allows users to obtain a set of event logs by simulating a MAS model w.r.t. to interaction constraints.

We have developed five use cases for evaluating the proposed simulation approach. In each case we have generated event logs with 5000 traces. For each case we provide a “filtered” version of event log with respect to interacting actions, s.t. it is clear whether the interface is observed exactly. We have used Disco² to visualize generated event logs. Insignificant parts of agent nets are shown by shaded ovals.

a) *Sequencing*: Consider a system with three interacting agents (see Fig. 3). Each agent always executes one action. We have simulated it with respect to the interface $\mathcal{I} = A \triangleleft B \wedge B \triangleleft C$. Intuitively, in this case each agent prepares resources needed for the other agent.

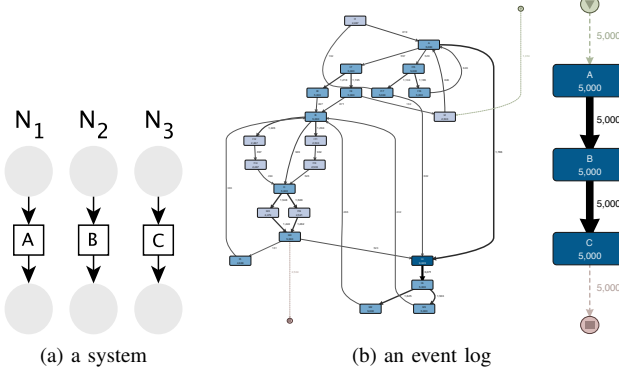


Fig. 3. Sequential interaction

b) *Conditional Sequencing*: As opposed to sequencing, conditional sequencing allows for several execution options. In this case, a system consists of two agents, one of which has alternative branches (see Fig. 4). Interface for the conditional sequencing is as follows: $\mathcal{I} = A \triangleleft C \vee C \triangleleft B$.

c) *Alternative interaction*: The alternative interaction implies that one of two interacting agents influences the choice done by the other agents. A system consists of two interacting agents both having alternative branches (see Fig. 5). An interface formula for this case is as follows: $\mathcal{I} = A \triangleleft C \vee B \triangleleft D$.

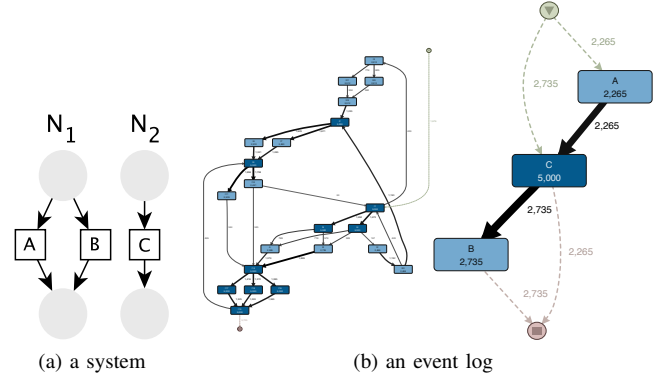


Fig. 4. Sequential interaction with options

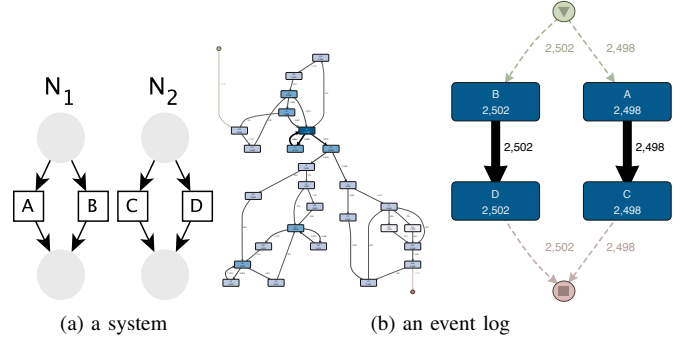


Fig. 5. Alternative interaction

d) *Interaction using negative constraints*: Assume we have a system of two interacting agents with two alternative branches as shown in Fig. 5a. The result of simulating this system with respect to the interface $\mathcal{I} = A \triangleleft C$ is shown in Fig. 6. It is clear from the simulation result that C is never preceded by A . Intuitively, negative constraints allow for a more compact way of interface construction.

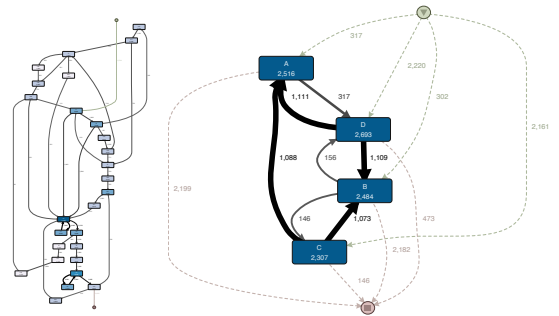


Fig. 6. Interaction using negative constraints: an event log

e) *Complex interaction*: In this case, we show several ways of interaction among three different agents (see Fig. 7a). For convenience, we have filtered the obtained log in two ways (see Fig. 8). We have used the following interface formula (given in a conjunctive normal form for convenience of reader): $\mathcal{I} = (B \triangleleft A) \wedge (H \triangleleft C) \wedge (D \triangleleft F \vee E \triangleleft G)$.

¹ProM 6 Framework page: <http://www.promtools.org>

²Fluxicon Disco page: <https://fluxicon.com/disco/>

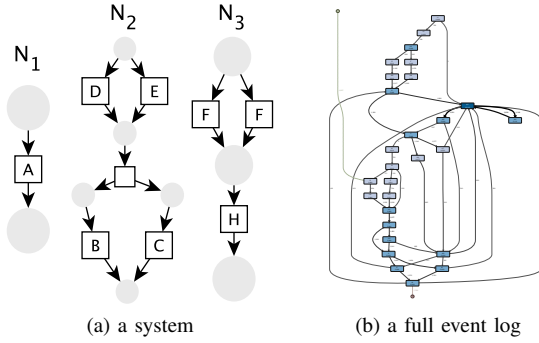


Fig. 7. Complex interaction

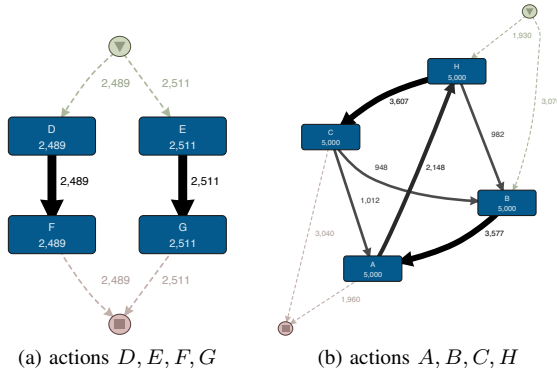


Fig. 8. Complex interaction: filtered event logs

VI. CONCLUSION

We have proposed the new approach to model and simulate multi-agent systems using Petri nets. Independent agents are modeled using a set of labeled workflow nets, and their interaction is described using a declarative interface. The interface is constructed as a logic formula over atomic constraints describing the order of internal agent actions. This study have considered only acyclic agent interactions described by two kinds of atomic constraints, s.t. interacting activities are implemented only once. If cyclic interactions are allowed, more subtle relations on interacting activities are needed to express such constraints as “each B should be preceded by A ” or “at least one B should be preceded by A ”. This is a subject for further research.

An algorithm for simulating process models of multi-agent systems with respect to the interface has been constructed. We have implemented the algorithm within ProM 6 plug-in GENA and have evaluated it using different cases of agent interactions. The experiment results show how to use our approach for describing agent interactions.

VII. ACKNOWLEDGMENTS

This work is supported by the Basic Research Program at the National Research University Higher School of Economics and Russian Foundation for Basic Research, project No. 16-01-00546.

REFERENCES

[1] W. M. P. van der Aalst, *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[2] C. W. Günther and W. M. P. van der Aalst, “Fuzzy mining: Adaptive process simplification based on multi-perspective metrics,” in *BPM 2007*. Springer, Heidelberg, 2007, pp. 328–343.

[3] J. M. E. M. van der Werf, B. F. van Dongen *et al.*, “Process discovery using integer linear programming,” *Fundam. Inform.*, vol. 94, no. 3–4, pp. 387–412, 2009.

[4] A. Weijters and J. Ribeiro, “Flexible Heuristics Miner (FHM),” in *IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011)*. IEEE, 2011, pp. 310–317.

[5] A. A. Kalenkova, I. A. Lomazova, and W. M. P. van der Aalst, “Process model discovery: A method based on transition system decomposition,” in *ICATPN 2014*, ser. LNCS, vol. 8489. Springer, 2014, pp. 71–90.

[6] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, “Scalable process discovery with guarantees,” in *BPMDS/EMMSAD*, ser. LNBIP, vol. 214. Springer, 2015, pp. 85–101.

[7] A. K. Begicheva and I. A. Lomazova, “Discovering high-level process models from event logs,” *Modeling and Analysis of Information Systems*, vol. 24, no. 2, pp. 125–140, 2017.

[8] A. Augusto, R. Conforti *et al.*, “Automated discovery of process models from event logs: Review and benchmark,” *CoRR*, vol. abs/1705.02288, 2017.

[9] V. A. Rubin, A. A. Mitsyuk *et al.*, “Process mining can be applied to software too!” in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, pp. 1–8.

[10] M. Leemans and W. M. P. van der Aalst, “Process mining in software systems: Discovering real-life business transactions and process models from distributed systems,” in *MODELS 2015*. IEEE, 2015, pp. 44–53.

[11] M. Leemans, W. M. P. van der Aalst, and M. van den Brand, “Recursion aware modeling and discovery for hierarchical software event log analysis (extended),” *CoRR*, vol. abs/1710.09323, 2017.

[12] C. Liu, B. van Dongen *et al.*, “Component behavior discovery from software execution data,” in *SSCI 2016*. IEEE, 2016, pp. 1–8.

[13] K. V. Davydova and S. A. Shershakov, “Mining hybrid UML models from event logs of SOA systems,” *Proceedings of the Institute for System Programming*, vol. 29, no. 4, pp. 155–174, 2017.

[14] “3TU: Big software on the run.” [Online]. Available: <http://www.3tu-bs.nl>

[15] L. Cabac and N. Denz, “Net components for the integration of process mining into agent-oriented software engineering,” in *Transactions on Petri Nets and Other Models of Concurrency I*, ser. LNCS, vol. 5100. Springer, Heidelberg, 2008, pp. 86–103.

[16] L. Cabac, N. Knaak *et al.*, “Analysis of multi-agent interactions with process mining techniques,” in *Multiagent System Technologies*, ser. LNCS, vol. 4196. Springer, Heidelberg, 2006, pp. 12–23.

[17] R. Nesterov and I. Lomazova, “Using interface patterns for compositional discovery of distributed system models,” *Proceedings of the Institute for System Programming*, vol. 29, no. 4, pp. 21–38, 2017.

[18] R. A. Nesterov and I. A. Lomazova, “Compositional process model synthesis based on interface patterns,” in *TMPA 2017*, ser. CCIS, vol. 779. Springer, Heidelberg, 2018, pp. 151–162.

[19] A. Burattin, “PLG2: Multiperspective process randomization with online and offline simulations,” in *Proceedings of the BPM Demo Track 2016*. CEUR Workshop Proceedings, 2016, pp. 1–6.

[20] T. Jouck and B. Depair, “PTandLogGenerator: A generator for artificial event data,” in *Proceedings of the BPM Demo Track 2016*. CEUR Workshop Proceedings, 2016, pp. 23–27.

[21] Z. Yan, R. M. Dijkman, and P. Grefen, “Generating process model collections,” *Software and System Modeling*, vol. 16, no. 4, pp. 979–995, 2017.

[22] I. S. Shugurov and A. A. Mitsyuk, “Generation of a Set of Event Logs with Noise,” in *Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014)*, 2014, pp. 88–95.

[23] A. A. Mitsyuk, I. S. Shugurov *et al.*, “Generating event logs for high-level process models,” *Simulation Modelling Practice and Theory*, vol. 74, pp. 1–16, 2017.

[24] A. K. A. d. Medeiros and C. W. Günther, “Process Mining: Using CPN Tools to Create Test Logs for Mining Algorithms,” in *Proceedings of CPN 2005*, ser. DAIMI, vol. 576. University of Aarhus, 2005, pp. 177–190.

[25] C. Di Ciccio, M. L. Bernardi *et al.*, “Generating event logs through the simulation of Declare models,” in *Enterprise and Organizational Modeling and Simulation*. Springer, 2015, pp. 20–36.

Human readable Extended Finite State Machine format

1st Alexander Nikitin

Computer science department

Higher school of economics

Moscow, Russia

alexnikleo@gmail.com

Abstract—Extended finite state machines (EFSMs) are widely used in formal testing and formal verification of software and hardware systems. EFSMs have a number of differences with classical finite state machines FSMs. For instance, EFSMs can have a predicate and/or update functions on every transition, and a set of context variables, and thus, it helps to describe more complex systems. However, despite the fact that there are many scientific works which use EFSMs, there is no unified format for EFSMs' description. The aim of this paper is to propose such format and to show its' advantages and disadvantages in applications to software formal testing.

Index Terms—formal testing, testing, extended finite state machine, finite state machine, parser

I. INTRODUCTION

The number of critical systems rapidly increases during last twenty years. We define critical systems as the systems which have to always operate safe [1]. Critical systems include aircraft software, connection protocols, medical equipment, software for atomic energy stations, etc. Such systems have to be thoroughly tested and verified to guarantee fault tolerance. Many modern methods of testing and verification of such systems use the model of Extended Finite State Machine (EFSM) as an underlying model. For instance, generation of the test sequences [2] and formal verification [3] can be done using the EFSM system specification.

Because of the large number of researchers which are using EFSMs it is important to have an unique readable format together with corresponding tools for the EFSMs' analysis, which can be used among researchers. Consequently it can result in the repository of EFSMs which can be reused by other researchers. Previously, there were a few works with the aim to parse an EFSM from HDL description, for example [4], but obviously this approach could not be easily generalized to all applications of the EFSMs. So the development of such format could increase reproducibility of researches and productivity of every single researcher. The main properties of the such format should be:

- unification (any existing format could be replaced with such format);
- compactness (ESFM representation should be short and do not take much disk space);
- human readability (one should be able to edit EFSMs represented in this format without any special tools, just with a text editor);

- existence of a fast parser (format could be parsed in the satisfactory time);
- scalability (should work with large EFSMs, that have the number of transitions up to 10^8);
- tools (should exist open source tools for parsing, analysis and visual representation).

The format proposed in this article possesses all these properties (besides open source tools, but creation of them is already in progress).

The structure of the paper is as follows. In Section II, Extended Finite State Machines (EFSMs) are defined and the section contains main theoretical preliminaries. Section III contains the description and specification of proposed format. Section IV describes a parser and has an example of the parser architecture. In Section V some experimental results are presented and Section VI concludes the paper.

II. PRELIMINARIES

In this paper we will use slightly modified definitions from [2].

Extended Finite State Machine (EFSM) is a triple $EFSM = (S, T, initial)$. Context variable are hidden variables, which are used in EFSMs. Set of the possible values of the context variables is D_v . A context vector $v \in D_v$ is called a context of an EFSM.

A configuration of an EFSM is a pair (state, values of context variables).

S is a set of states, T is a set of transitions and $initial$ is an initial configuration of the EFSM. Which includes initial state and initial values of the context variables: $initial \in S, D_v$, where D_v is the set of possible context variables. Often EFSMs are defined without $initial$, we use $initial$ for convenience. Equivalence of these definitions is obvious.

T is a set of transitions $t \in T$ where $t = (from, to, P, U, x, y)$.

Let D_{inp-x} be the set of possible input vectors, D_{out} be the set of possible output vectors. X and Y are the sets of possible inputs and possible outputs respectively.

Then,

$from, to \in S$ – the initial and final states of a given transition;

$x \in X$ – input;

$y \in Y$ – output;

$P \in D_{inp-x} \times D_v \rightarrow \{True, False\}$ – predicate, sometimes

it is called guard;

$U \in D_{inp-x} \times D_v \rightarrow D_v$ – update function.

Given an input $x \in X$ and a vector $\rho \in D_{inp-x}$, the pair (x, ρ) is called a parameterized input. Parameterized input sequence is a sequence of parameterized and non-parameterized inputs. Given a parameterized input sequence of the EFSM we can calculate the corresponding parameterized output sequence by simulating the behavior of the EFSM under the input sequence starting from the initial configuration. It can be done by simple traversal of the EFSM.

We can see an example of the simple EFSM in Fig. 1. There are three transitions and three states, transitions are represented in the proposed format (see Section III). In the top left corner initial values of the EFSM's context variables are presented.

III. FORMAT SPECIFICATION

The format can be described with the pseudocode as follows:

```
states: v1, v2, ..., vn;
init: vi;
context_vars: {
  c_vars.x = 11;
  c_vars.y = 0;
};
transitions: [
{
  from: v1;
  to: v2;
  predicate: {
    return input.a == 1 and c_vars.x == 2;
  }
  update:
    c_vars.x = c_vars.x + 10;
},
...
];
```

Listing 1. Pseudocode to describe .efsm file.

We now look at the config fields more precisely:
states is a list of the comma-separated state names;
init is a the initial state;
context_vars are initial values of the context variables;
transitions are a list of transitions, every transition is described inside braces;
Transitions are described by the following fields:
from - the initial state of a given transition;
to - the final state of a given transition;
predicate - a function written in any programming language. For describing predicates, the current prototype uses the Python syntax. Full specification can be found in the Python's standard documentation [5] with some additions: it also can use input.{name of input variable} format to reference input variables and c_vars.{name of context variable} format to reference output variables. It must return a boolean value as the result;

update - similar to predicate it is a function, but the aim of this method is to update context variables. The value of the function is in the range of a corresponding context variable or an output parameter.

Formally the format can be described with Backus-Naur form (BNF) [6] Our grammar is shown in Listing 2.

```
possible_chars = 'A-Za-z0-9-_.~%+*/' ()|'
EXPR_DELIM = ;
LIST_DELIM = ,
state = possible_chars*
context_vars = c_var.(word)

predicate_expr = PYTHON_CODE;
update_expr = PYTHON_CODE;

transition_from = from: integer;
transition_to = to: integer;
transition_input = input: {};
transition_output = output: {};
transition_predicate = predicate: predicate_expr
transition_update = update: delimitedList(update_expr)
transition = {transition_from + transition_to +
              transition_input + transition_output +
              transition_predicate + transition_update};

transition = "transitions: [" delimitedList(transition) "];"

states = "states:" + delimitedList(state, delim=LIST_DELIM) + EXPR_DELIM
init = "init: " + state + EXPR_DELIM
grammar = states + init + transition
```

Listing 2. BNF of .efsm file.

It uses macro delimitedList in the pyparsing syntax [7]. Delimited list used to be short, but obviously it also can be defined using pure Backus-Naur form (BNF). PYTHON_CODE is the Backus-Naur form (BNF) of the python programming language [5], we do not expand it because of the large size.

IV. PARSER ARCHITECTURE

When constructing the EFSM system specification it is very desirable to have a human readable format. However, this format is not very efficient when performing some operations over EFSMs such as for example, modeling an EFSM behavior under a given parameterized input sequence. For these reasons, we need a parser and its architecture is shown in Section III Parser architecture can be shown in Fig. 2. It consists of the two main components: a lexical parser and a syntactic parser. Lexical parser takes .efsm file described in format shown in Listing 2 and returns a stream of tokens. Stream of tokens is given as an input to the syntactic parser which uses it to produce an EFSM object, which than can be used to work with EFSM (generate tests, perform verification, etc.). This scheme is often used in compilers and translators [8] and can be used for parsers which are written in different programming languages, for example, Flex and Bison in C++, ANTLR in Java, etc.

We use pyparsing [7] python library for the lexical parser in the prototype and pure python implementation of the syntactic parser.

V. EXPERIMENTAL RESULTS

A. Experimental setup

We conducted a number of experiments to check the performance of the created parser and to create benchmarks for the further works. We performed the series of experiments measuring the working runtime. Experiments were performed using a computer with the processor: 2,7 GHz Intel Core i5, RAM: 16 Gb 1867 MHZ DDR3, time was measured with Linux utility "time" while the memory consumption was analyzed with python library psutil [9].

Fig. 1. Example of a simple EFSM.

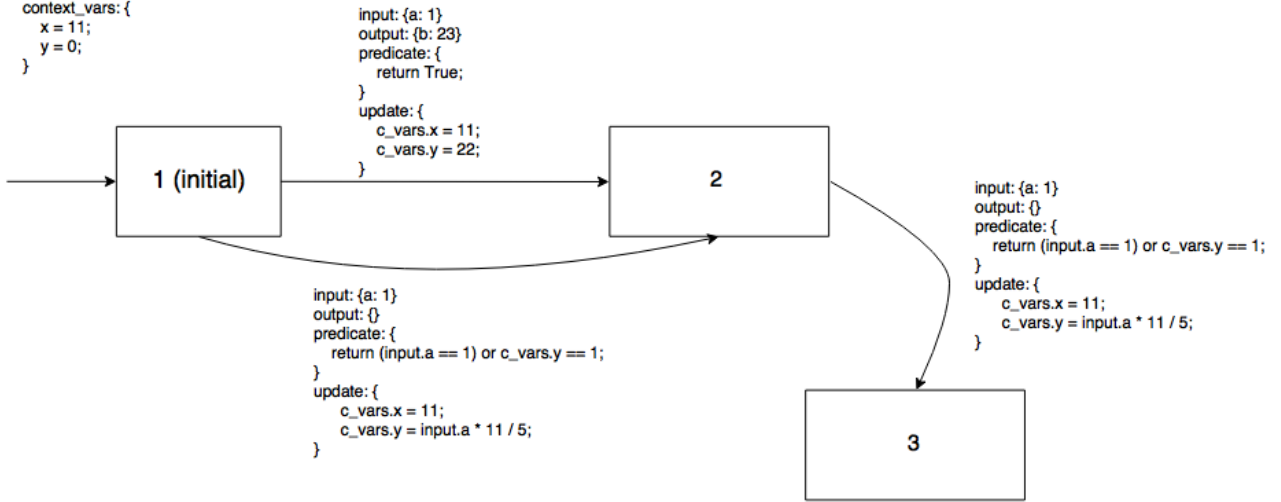
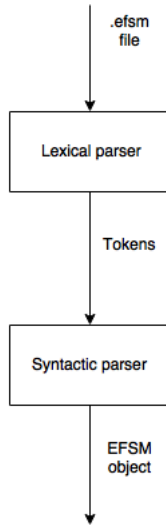


Fig. 2. EFSM parser scheme.



$\mu \pm \sigma$ where μ is a mean and σ is a standard deviation. Timings

TABLE I
TIME OF THE PARSER EXECUTION.

| Number of transitions | Execution time, seconds |
|-----------------------|-------------------------|
| 10 | 0.072 ± 0.004 |
| 100 | 0.11 ± 0.004 |
| 1000 | 0.58 ± 0.004 |
| 10000 | 5.396 ± 1.147 |
| 100000 | 45.966 ± 3.779 |
| 1000000 | 521.96 ± 14.840 |

of the parser runtime trials are presented in Table I.

We also tested the parser on EFSM representations of simplified real-world systems, e.g. fast real number power algorithm, ATM specification and TCP specification. The parser works several seconds and the size of the specifications for these systems is less than dozen of Kb. It shows the compactness and practical usefulness of the given format.

TABLE II
MEMORY CONSUMPTION OF THE PARSER.

| Number of transitions | Memory consumption, Mb |
|-----------------------|------------------------|
| 10 | 10 |
| 100 | 11 |
| 1000 | 15 |
| 10000 | 64 |
| 100000 | 550 |
| 1000000 | 4510 |

B. Time of the parser execution

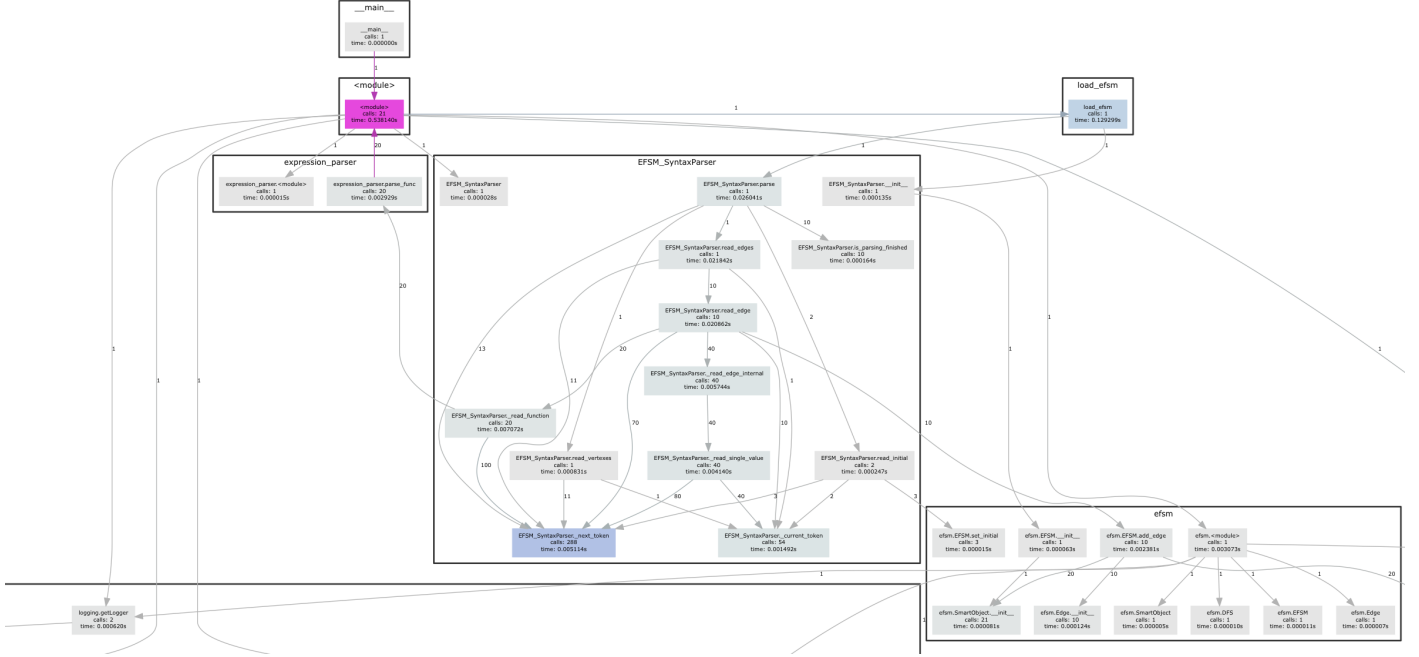
In order to make a good benchmark we have to choose type of the EFSMs for benchmark trials. We drew call-graph and profiled source code with cProfile to find possible bottlenecks. As we can see in Fig. 3, the most complex operation is the edges' parsing. Knowing that, we decided to create a set of benchmarks of the EFSMs with $N+1$ states and N transitions with transitions from the state i to the state $i+1$. Asymptotic of the parser is $O(n)$ but we also have to check how it will perform in the computer to measure runtime.

For each number of transitions N we conducted 5 experiments to calculate the mean and standard deviation to get rid of the systematic error. All results are presented in the format

C. Memory consumption

Also we studied the memory consumption of the parser. We used the same experimental setup as in the runtime experiments. Detailed results are presented in Table II. The parser consumes relatively much memory because of the large size of the standard python data structures (e.g. strings and

Fig. 3. Call graph of EFSM parser run.



tuples), function objects and SmartObjects on the every edge (SmartObject is a special object type developed in the parser to allow addressing context variables and input symbols from the predicate and update functions).

The storage of functions can be optimized by storing text representations of some or every function by finding tradeoff between performance and memory efficiency. SmartObjects could be optimized with usage C structures or with a more clever parsing of the python code. The parser's memory optimization needs additional research and is left for the future works. If memory consumption is an issue in the practical application of the given format we recommend to use C implementation of the parser.

VI. CONCLUSIONS

In this paper, we described the EFSM format that can be used in a number of applications. The format meets the requirements indicated in Section I. We presented the parser prototype to process EFSMs which are written in this format and we prepared benchmarks to test EFSM parsers and profiled the prototype. Also we ran the parser using some tests and found out the limitations of the parser. The parser works very fast and needs a relatively small amount of memory and this illustrates the practical usefulness of the proposed format. The performance of the parser can be improved and methods to improve it are also mentioned. Main directions of the future research include improvement of the parser's performance and check of the proposed format on the large real-world systems. The predicates and update functions are written in python now, but it could be easily replaced by any other function description (other programming languages

or any formal function description). We really hope that the EFSM repository could also be created in the nearest future using the proposed format.

REFERENCES

- [1] Ian Sommerville, "Software Engineering (7th Edition)", 2004
- [2] A. Petrenko, S. Boroday, R. Groz, "Confirming configurations in EFSM testing", IEEE Transactions on Software Engineering (Volume: 30, Issue: 1, Jan. 2004)
- [3] Giuseppe Di Guglielmo, Franco Fummi, Graziano Pravadelli, Stefano Soffia, Marco Roveri, "Semi-formal functional verification by EFSM traversing via NuSMV", High Level Design Validation and Test Workshop (HLDVT), 2010 IEEE International
- [4] S.A. Smolov, A.S. Kamkin, A method of extended finite state machines construction from HDL descriptions based on static analysis of source code, St. Petersburg State Polytechnical University, 2015 - mathnet.ru
- [5] Full specification of python source code <https://docs.python.org/2/reference/grammar.html>
- [6] Backus, J. W. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference", Proceedings of the International Conference on Information Processing. UNESCO, 1959, pp. 125-132.
- [7] Pyparsing documentation <http://pyparsing.wikispaces.com/Documentation> for the documentation
- [8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman Compilers: Principles, Techniques, and Tools (2nd Edition), 2007
- [9] Documentation and source code of python library psutil, <https://github.com/giampaolo/psutil>

Criteria for software to safety-critical complex certifiable systems development

Natalia Gorelits

Department 2100: Advanced Systems
and Avionics Integration
State Research Institute of Aviation
Systems (GosNIIAS)
Moscow, Russia
nkgorelits@2100.gosniias.ru

Aleksandra Gukova

Department 2100: Advanced Systems
and Avionics Integration
State Research Institute of Aviation
Systems (GosNIIAS)
Moscow, Russia
asgukova@2100.gosniias.ru

Evgeniy Peskov

Department 2100: Advanced Systems
and Avionics Integration
State Research Institute of Aviation
Systems (GosNIIAS)
Moscow, Russia
evpeskov@2100.gosniias.ru

Abstract— Nowadays there is an actual problem in aviation industry – how to make the development of complex safety-critical systems certifiable according to international and domestic standards and regulations like DO-178C, DO-254, ARP 4754A, ARP 4761 etc. In the article configuration management process from the development lifecycle of DO-178C is considered as the main source of criteria for the development tool selection. Selected criteria can be applied to software tool which supports entire development lifecycle of aviation software, as well as to software tools supporting some individual lifecycle processes. As example of criteria using one of the most widely known in industry software tool for requirements development and management was analyzed for compliance with the chosen criteria.

Keywords — DO-178C, Qualification requirements 178C, software development, software analysis, software choosing, certifiable systems, complex systems, complex systems development, avionics, on-board equipment, lifecycle processes, lifecycle, configuration management, system engineering

I. INTRODUCTION

This research was inspired by acquaintance and very productive work communication with untimely gone Michael Saburov. Michael Saburov participated in development of Russian analogs of certification standards and regulations DO-178B [1], DO-178C [2], DO-254 [3] and DO-330 [4]. Also Michael Saburov participated in implementation of processes from these regulation documents in several industry enterprises. Michael Saburov took an active part in formation of the concepts of research described in this article [5]. All results and experience gained by us during work on this research are dedicated to Michael Saburov.

Development and the following certification of complex safety-critical systems in compliance recommendations of regulation documents DO-178C, DO-254, ARP 4754A [6], ARP 4761 [7] is an actual task and a big challenge for modern Russian aviation industry.

Today among the software announced by its developers as supporting lifecycle of complex systems development a huge number of products are presented to allow software development in accordance with international quality standards.

Nevertheless at the moment assessment of the capabilities of each tool (or often it will be the whole product line of

expensive tools) and making a reasonable choice is rather difficult problem.

Big quantity of existing software tools and systems positioned by developers as tools which support lifecycle processes of complex systems development don't have well-founded assessments.

Assessments and reviews about such software, based on experience of practical usage in industry projects, are very important – software market proposes a lot of software tools and systems made by Russian and foreign developers. So that's why industrial enterprises have to make difficult choice of software tools for development and the following certification of their critical-safety systems.

It is difficult to choose instrumental environment for support the entire development lifecycle – unfortunately universal multipurpose tool, which would satisfy the requirements of all standards of all industries, doesn't exist yet.

In general most of the enterprises use separate tool for support and automate each process of development lifecycle (like requirements development process, configuration management processes, verification etc.). The situation is complicated because often all or the most parts of such software suite have different manufacturers. If the project have big set of weakly integrated software, then product development becomes more and more complex both in atomic tasks of individual specialist and in global meaning of the whole project – labor intensity increases.

The organization of development landscape as a bunch of software tools entails difficulties with tools integration, training costs, implementation costs, purchase of licenses. All these changes increase the amount of resources which are needed for successful completion of the processes – human resources, financial and time resources. In this case reaching project goals, formulated before the beginning of work, become more and more difficult task.

In conditions of State program of import substitution [8] software tools and systems made by Russian developers cause big interests. But usage experience of Russian software and consequently number and reliability of assessments according to requirements listed above standards and regulations are not big enough.

In this article we tried to understand and present what mechanisms and features software tool should have to be useful to simplify and systematize development of certifiable aviation software. This article is a part of series of materials about aviation standards research in context of choosing software tools for certifiable aviation software development [9].

II. DO-178C PROCESSES AND THE ROLE OF CONFIGURATION MANAGEMENT PROCESS AMONG THEM

Russian analogue of DO-178C - Qualification requirements 178C [10] – regulates processes of certifiable development of aviation software. The heading of Russian document contains important words – “Requirements to the software of on-board equipment and systems at certification of aircraft”. These words uniquely determine goals of recommendations, specified in the document.

Certifiability of product – significant property, because the purpose of most developments is the following release of end-product the on relevant market. In the context of aviation systems certifiability means that aircraft with system included will receive type certificate [11].

Under certifiability assurance we mean the implementation of the development processes in specific way –

- all necessary for certification activities are performed,
- all necessary for certification objectives are achieved,
- all necessary data is collected about development process and its result,
- this data is stored and processed in such a way that certification authority could receive any data at any stage of project in order to examine the data and to trace the history of their interactions and relationships.

Activities and objectives to airborne systems and equipment development are described in document DO-178C (Russian analogue – qualification requirements 178C). DO-178C provides instructive materials and guidance to create airborne systems and equipment. Implementation of activities and objectives achievement listed in DO-178C give a chance to get in the end the result which **performs its intended function with a level of confidence in safety that complies with airworthiness requirements**.

DO-178C describes a set of development lifecycle processes for aviation systems and equipment. DO-178C divides processes of the development lifecycle to three groups. The first group includes only one process – **software planning process**. The second group called **software development processes** includes four processes – software requirements process, software design process, software coding process and integration process. The third group consists of four **integral processes** – software verification process, software configuration management process, software quality assurance process, certification liaison process.

During software development processes directly creation of software of aviation systems takes place along with all previous and accompanying it measures for the design, coding, integration etc. The main result of development processes is the

executable object code and its associated additional data are produces and loaded into the target hardware for further integration. This result is necessary to be achieved having carried out all the measures described in qualification requirements.

Integral processes play a role of enabling processes (by analogy with enabling systems in the terms of System Engineering [12]) - created and edited during development processes data is stored and processes through mechanisms and activities of configuration management process, required reviews and analyses are made in the verification process and so on. Data – development lifecycle artifacts or configuration items – may be requirements with different levels of details, software architecture, source code and executable object code and different protocols, problem reports, and many other results of activities.

Explanation the importance of integral processes implementation is very simple – otherwise it is very difficult almost impossible to collect necessary for certification data and to control the development process. It means that it will be difficult to provide **necessary level of confidence in safety that complies with airworthiness requirements**.

Each of integral processes has its own role and importance in the development lifecycle, it couldn't be ignored or partially abolished during lifecycle. Huge risks await developers who dare not comply integral processes - certification authority will not accept results obtained this way and will not give relying certificate. Also final product may contain errors and defects of varying degrees of critically. This situation will not allow achieving the required level of confidence in safety and quality of result in total, if the development process comes to an end with the release of the working result.

In modern world of computers and upcoming information technologies the whole software development lifecycle (and aviation software is not an exception) passes through software tools, information systems and therefore its databases and repositories. These software tools and information systems for all kinds of operations on data (creation, storage, editing etc.) must be evaluated for their sustainability and compliance with development according to certain standards and other regulation documents.

If perform analyze requirements to development product, which Qualification requirements 178C specifies and requires developer, becomes obvious that the most restrictions and requirements for software (in which aviation software will be developed) come from configuration management process. Activities of configuration management process provide operations with development lifecycle data, its storage, informational support to data exchange between other lifecycle processes, logging the history of changes etc.

In this research we chose configuration management process as the source of arguments or justifications for choosing of software tools on which certifiable aviation software will be developed. These substantiations are formulated in the form of criteria. Criteria can be applied to potentially interesting software tools and systems from the market and help with assessment and reasonable choice of

some of them. There will be described below how to apply selected criteria to the most widely used (worldwide and also in Russia) software for requirements development in the industry.

III. BASIC CRITERIA TO TOOL FROM CONFIGURATION MANAGEMENT PROCESS

Configuration management process in project must be performed in accordance with the document "Software Configuration Management Plan". Software Configuration Management Plan should be developed for each software development project during Software Planning Process if development corresponds to Qualification requirements 178C. In this document configuration management environment should be determined as well as configuration management process activities which will be performed during software development lifecycle.

Configuration management environment must support activities from section 7.2 of Qualification requirements 178C. The list of configuration management activities contains some process regulations (which restrain project members within the workflow) and requirements to the mechanisms of configuration management environment. It would be very useful if such mechanisms and methods will be implemented in software which will be used for development because not all of them could be replaced with some organizational regulations.

Configuration management plan contains some requirements to configuration management activities follow-up. As examples of these requirements can be listed: states of configuration items, workflows of problem reports and change requests, inspection procedures, baseline definition rules and rules of versioning configuration items, organizational restrictions, safety details etc. These requirements won't be considered in this article because its implementation can be realized regardless of the instrumental part of configuration management environment.

In this article we identified the basic principles and mechanisms (basic criteria) determined by configuration management environment and configuration management activities according Qualification requirements 178C.

First of all we would like to highlight single and unified storage for all lifecycle data as basic configuration management principle. It means that project should have unified configuration management system for registration, storage and delivery all software development lifecycle data.

Let's enumerate basic mechanisms of configuration management environment:

- Identification of configuration,
- Configuration status accounting,
- Change management and control,
- Traceability,
- Versioning,
- Registration of inconsistencies and corrective actions,
- Storage, retrieval and release.

Described mechanisms (further: criteria) are based on text Qualification requirements 178C and are advisory in nature. These criteria can be used as an additional informational source while choosing software tool for certifiable aviation software development.

Elements of the criteria list will be considered in more detail below.

1) Identification of configuration and its configuration items

Procedure of identification of the configuration item (and the whole configuration in general) includes assigning an identifier to the configuration item and registering it in the configuration management system. The identifier of configuration item is a designation uniquely distinguishes one configuration item from another. Identifier of configuration item could not be changed ever. Identifier of configuration together with its version makes Unique identifier of configuration item in a particular configuration. Version of configuration item will be described below in one of the criteria.

An example of attributes that we suppose useful for registration of configuration item:

- Configuration item identifier (doesn't change ever after registration),
- Mnemonics (designation which will help user identify configuration item),
- Configuration item name,
- Purpose of configuration item (type),
- Kind of configuration item (atomic, composite – configuration index),
- Version (number, sign if it is baseline or not),
- Data control category (Control Category 1 or Control Category 2),
- Link to the configuration item source.

Note: software lifecycle data can be classified to Data Control Category 1 or to Data Control Category 2 (section 7.3 of Qualification requirements 178C).

2) Configuration status accounting

Status accounting of developing software configuration should be conducted in order to provide the certification authority all necessary information (like configuration index, history of configuration etc.). That's why it is necessary to ensure that registration of the actions performed on the configuration units is automatic.

An example of data which we suppose to necessarily register when performing any action on the configuration item:

- Date and time of making changes to the configuration item,
- Number of version of the configuration item,

- User id – who made changes to configuration item or created version of configuration item,
- Status of version of configuration item including the history of this status changes,
- For configuration items from Control Category 1: link to the change request for this configuration item.

3) Versioning, baselines

Rules of naming and versioning for configuration items should be defined.

Note: for example, configuration item's version is denoted as an integer (1, 2, 3 etc.). New value of configuration item's version is obtained by increasing the value by 1. If it was 2, the next value will be 3.

Rules for baseline formation and baseline appointment mechanism should be defined. Also restrictions on the baseline's modification should be defined.

Note: baseline is approved and registered version of configuration item which will be used as basic for further development. Baseline can consist of one or several configuration items.

4) Configuration items traceability

Traceability requirements and mechanisms should be defined for link different types of configuration items and related data. Configuration items can be connected with each other, also with reason of creation (source), with dependent items, with history of configuration item's changes etc.

Note: as example of connections we may mention links between low level requirements with its parent high level requirements, low level requirements with executable object code, problem report with configuration item, problem report with change request and with task for making approved changes etc.

Configuration items traceability is very important in the context of developing software certification. It is necessary for configuration items to trace links with source of its creation, with maked to configuration items changes and with reason to making changes etc.

Traceability of links should work in both directions. Changes in configuration items should trace to sources of changes (for example to change request, which in its turn refers to parent problem report) and back.

It is always useful for users to analyze some visualized view of data. As a variant of useful and intuitive view of links and traces may be a traceability matrix. Traceability matrix shows how configuration items are connected to each other and their relations type is displayed. Type of relations between configuration items can be presented both in simple form with only displaying link presence or absence, and in the various types of links and communication.

Table 1 illustrates an example of configuration item's baseline formation.

TABLE I. AN EXAMPLE OF TRACEABILITY MATRIX: LINKS BETWEEN CONFIGURATION ITEMS

| Configuration items | CI1 | CI2 | CI3 | CI4 | ... |
|--|-----|-----|-----|-----|-----|
| CI1 | | X | ↕ | | |
| CI2 | X | | | ↓ | |
| CI3 | ↕ | | | | |
| CI4 | | → | | | |
| ... | | | | | |
| ■ – not applicable X – connection exists ↓, ↕, → – certain type of connection exists | | | | | |

5) Change management and control

The change management of the configuration items must be implemented. Change management activities are responsible for the reaction to recording, evaluating, solving problems through the whole lifecycle of each configuration item.

Any change of configuration item should only be done by creating a new version of changing configuration item. However all previous versions should remain unchanged. Previous versions should be stored in repository and be accessible.

Changing of configuration items from Control category 1 is possible only through special procedure of change management. Problem report should be created and approved, detailed change requested and tasks should be created from this problem report. Changes to configuration items from change request should be also approved and only then changes may be applied to configuration items. All related information about changes must be stored forever – who, when, for what reason have changed that version of configuration item. Changes to configuration items with Control category 2 don't require complex procedure with approvals and reviews of changes.

6) Registration of inconsistencies and corrective actions

Once inconsistencies or defects are detected, it is necessary to determine procedure and mechanisms of its registration. Also corrective actions should be established, impact analysis of the proposed changes should be done and making of the approved changes to configuration item should be strictly controlled.

Any project member who discovered an inconsistency or defect or any other type of error, should be able to write it in special configuration item – problem report.

An example of attributes which we suppose to necessarily register when registering a problem report for any configuration item:

- Link to configuration item – source of detected inconsistencies,
- Link to index of configuration which includes configuration item with inconsistency or to process or workflow if inconsistency is more global,
- Inconsistence description,

- Problem report's author id,
- Steps to reproduce the problem,
- Problem report state,
- Link to corrective actions (for example: change request).

An example of attributes which we suppose to necessarily register when registering a corrective action for any problem report (for example: change request):

- Link to problem report (change request source),
- Link to configuration items in which it is necessary to make changes,
- Impact analysis of proposed changes to the rest configuration items of lifecycle data.

7) Storage, retrieval and release

Method and proof of data integrity should be determined during its storage and retrieval from backups. Rights to release data should also be defined. Tools for creation, retrieval and integrity control of backups should be implemented according to chosen method.

Note: the need for backup creation can be both for the entire repository and for a separate development project or for separate configuration.

The realization of instrumental support for the creation, retrieval and data integrity control is very important and in demand because it allows to minimize time costs for these procedures and to reduce the risk of data distortion or loss.

Note: using of a checksum mechanisms for backups creation may be a good example of data integrity control realization.

IV. CONFIGURATION MANAGEMENT TOOLS, ANALYSIS

The experience of cooperation with Russian developers of avionics system demonstrates that most of them try to create on-board software in compliance with the requirements of the document Qualification requirements 178B/C and then certify their software products.

At the same time there are situations when the software development process is produced without detailed requirements (in fact without requirements at all - only high-level technical specification are used), without configuration management, without reviews or inspections. Software testing is conducted, but unfortunately its completeness can be insufficient by reason of the absence or incompleteness of requirements.

Realizing their unpreparedness for further certification without using of specialized software, aviation enterprises are implementing various tools. An example of such tools can be IBM Rational DOORS, IBM Rational Change + Synergy, IBM Rational Team Concert, Siemens Team Center Requirements, LDRA and others. In this case often overlooked that without understanding the processes (and not having the described processes on a paper at least) it is almost impossible to get the effect of the implementation of the tool.

It is necessary to apply the certification process with a complex approach to achieve the best result. It means - to develop the processes, to provide their support by tools, to develop plans and standards (Plan for Software Aspects of Certification, Software Development Plan, Software Verification Plan, Software Configuration Management Plan, Software Quality Assurance Plan; Software Design Standards, Software Code Standards, Software Requirements Standards) and to conduct development in full compliance with these plans and standards.

Often enterprise of the aviation industry implement only tool for writing and storage requirements. Typically, this tool has minimal change management capabilities. Developers try to manage requirements ignoring or paying low attention to the configuration management process – this approach is fundamentally incorrect.

Below we put a list of the most widely used tools to support the software development lifecycle, implemented in Russian aviation enterprises.

To support requirements management processes are often used: Microsoft Excel / Word, IBM Rational DOORS, Siemens TeamCenter Requirements Management (mainly in those enterprises where Siemens TeamCenter PLM was previously implemented in the design department) and even more rare - 3SL Cradle.

Due to the State program of import substitution, products of Russian developers arouse great interest. Among the most ambitious it is possible to highlight product which supports the entire development lifecycle of systems - Devprom.

To support lifecycle data change management processes are often used: IBM Rational Change + Synergy (tools are not supported by the vendor, but are still in use in some enterprises), IBM Rational Team Concert, and the most popular project and task management tools - Redmine and Atlassian Jira.

In situation when the software product Redmine or Jira are used to manage changes to the lifecycle data, the integration between these tools is rather nominal – all tools supported development lifecycle work independently, links between change requests and requirements are fixed in a text file.

This approach does not contradict the principles of configuration management prescribed in Qualification requirements 178C, but not only doesn't simplify the development process, but also makes the process management even more difficult (dependence on the human factor, the inability to track changes (the absence of a change marker), the lack of quick switch from a change request to the changed data, etc.)).

To support configuration management processes are often used: GitHub - the most popular and freely distributed tool among code developers and SVN (Subversion) - a traditionally used repository for file sharing in enterprises in Russia (also distributed under the conditionally free Apache license).

The functionality of these tools when it used as configuration management systems does not allow you to fully

support all activities of the configuration management section 7.2 of Qualification requirements 178C. Moreover, the use of all the functionality of this software may be considered as a violation of some of them. It is almost impossible to restrict the functionality of tools that are useful to traditional code developers in order to comply with the process specified in the Configuration Management Plan.

For example, GitHub does not store intermediate versions when you merge code branches (or other files when you use this tool as a configuration management environment) and you cannot track changes that precede the merge.

Quote from DO-178C (section 7.2.4 e): “Throughout the change activity, software life cycle data affected by the change should be updated and **records should be maintained** for the change control activity”.

For the analysis for compliance with the criteria described in the previous section, we present the summarized results of the requirements management tool IBM Rational DOORS use in State Research Institute of Aviation Systems (GosNIIAS) and the results of the analysis of the entire IBM Rational product line for lifecycle management [13].

We can analyze requirements management tools for conformity by Configuration Management process criteria, because the requirement is one type of configuration items and recommendation of section 7.2 of Qualification requirements 178C about its storage and handling must be observed.

To evaluate the criteria, the following values (weight) were selected:

- 0 – criteria is not supported;
- 0.5 – criteria is partially supported;
- 0.75 – criteria is supported through tool configuration, adaptation or any integration;
- 1 – criteria is fully supported.

The analysis results are shown in the figure below on Fig.1.

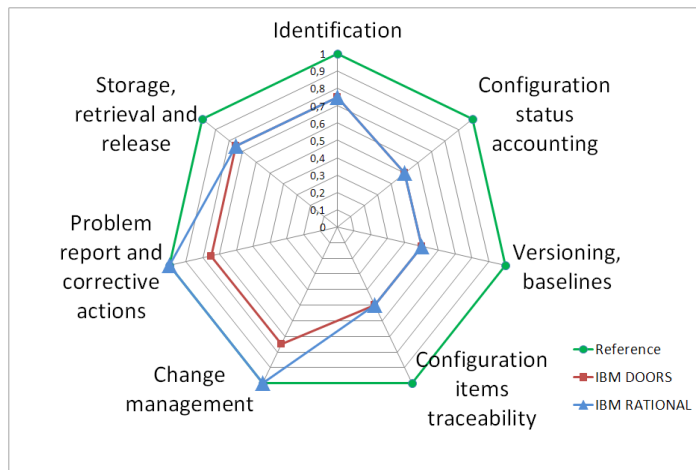


Fig. 1. Tools analysis

V. CONCLUSION

Configuration management process – is the main source of criteria for choosing the tools which support aviation software development lifecycle. Configuration management process acts as unifying “input-output bus” for all lifecycle data. Therefore tools with support of the software development lifecycle should focus on the mechanisms, embedded in the configuration management process, in order to be able to interact closely (be integrated). Such a close relationship (integration) through the configuration management process can significantly help with the development process, provide a predictable (and positive, if the tool was chosen correctly) result of aviation software development and help with preparing to the certification. It is important to note, that the purchase of the software tools and instruments doesn’t ensure success in passing the certification – methodological support is also needed.

The task to select software tools for development lifecycle support is not easy, because it is rather difficult to determine in advance whether all requirements of chosen for this project lifecycle process will be supported by software tool, system or a set of tools. Analysis of configuration management process and selecting criteria from it to tools allows to define the boundaries of necessary for the project systems and tools. Analysis gives as result formulated requirements to the tool, which can be applied for choosing and buying suitable tool or in case of independent development such instrumental environment. In case of buying these requirements and criteria will help to choose exactly that product whose functions are necessary and sufficient for development goals without spending a lot of money for buying disparate software tools of different manufacturers which will complicate the solution as a whole.

These conclusions are confirmed by the above analysis of one of the tools. Using of the set of tools extending the functional brings the environment closer to the reference state of configuration management process. Also there are difficulties: often the cost of licensing significantly increases (you have to buy additional tools), the time for installation, integration and implementation of the process increases, number of tools used in the project is growing and requires management efforts. As a result the total complexity of development increases.

REFERENCES

- [1] Software Considerations in Airborne Systems and Equipment Certification (RTCA DO-178B), 1992.
- [2] Software Considerations in Airborne Systems and Equipment Certification (RTCA DO-178C), 2011.
- [3] Design Assurance Guidance for Airborne Electronic Hardware (RTCA DO-254), 2000.
- [4] Software Tool Qualification Considerations (RTCA DO-330), 2011.
- [5] M.A. Saburov, “SCM-178C”, unpublished (in Russian)
- [6] Aerospace recommended practice. Guidelines for development civil aircraft and systems (SAE ARP 4754A), 2010
- [7] Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment (SAE ARP 4761), 1996
- [8] The Order of the Ministry of Industry and Trade of the Russian Federation of March 31, 2015 № 663 “About the approval of the

industry plan of actions for import substitution in branch of civil aircraft industry of the Russian Federation” (with changes and additions)

- [9] N. Gorelits, E. Peskov, “Requirements management as efficiency measure for software development in aviation industry”, Proceedings VIII International conference “IT-Standard 2017”. Moscow, 2017, pp.105-113 (in Russian)
- [10] Qualification requirements part 178C, IAC, 2014 (in Russian)
- [11] M.A.Saburov, Yu.A.Solodelov, N.K.Gorelits, “Development of the certifiable avionics software by the example of JetOS real time operation system”, Proceedings of Third All-Rus. Scient.-Technical Konf. “Navigation, guidance and control aircraft”, Moscow, 2017, pp.241-243 (in Russian)
- [12] System engineering — System life cycle processes (ISO/IEC/IEEE 15288:2015), 2015
- [13] I.V.Koverninsky, A.V.Kan, V.B.Volkov, Yu.S.Popov, N.K.Gorelits, “Practical experience of software and system engineering approaches in requirements management for software development in aviation industry”, Proceedings ISP RAS 28(2), Moscow, 2016, pp.173-179

Formalizing Metamodel of Requirement Management System

Kildishev Denis Stepanovich
Ivannikov Institute for System
Programming of the RAS
Moscow, 109004, Russia
kildishev@ispras.ru

Khoroshilov Alexey Vladimirovich
Ivannikov Institute for System
Programming of the RAS
Moscow, 109004, Russia
khoroshilov@ispras.ru

Abstract—Requirements play an important role in the process of safety-critical software development. To achieve reasonable quality and cost ratio a tool support for requirements management is required. The paper presents a formal definition of a metamodel that is used as a basis of Requality requirements management tool. An experience of implementation of the metamodel is discussed.

Keywords—requirement, model, requirements management

I. INTRODUCTION

The development of the big and complex system is always a sophisticated task. It is ever more right for safety-critical systems where the cost of error is definitely high. This leads us to question on how to ensure that system is safe. One of solution for that is precise and accurate requirements' management.

Modern software development is based on requirements in different forms — from plain texts on natural language to some specific DSLs. The development of avionics systems tends to be based on some standards like ARINC. Those standards provide requirements in form of a structured document but this form may be not so suitable for the development of end system.

So from one side, we need to manage some requirements in form of semi-structured standards and from another, we must support more formalized specification for development and further activities [1]. In this case, we need a tool that can manage requirements in both forms. Moreover, we need to support the relations between those requirements and other development artefacts like tests and code and we have to support precise changes management.

The paper presents a formal definition of a metamodel that is used as a basis of Requality requirements management tool. Implementation details of support of the metamodel in the tool are discussed.

II. RELATED WORKS

The problem of requirements management is not a new one. This activity was known as very important for years. As an example we may cite some publication from 1997:

"The inability to produce complete, correct, and unambiguous software requirements is still considered the major cause of software failure today" [2].

But the requirements engineering task is still the subject of different investigations. One of them defines some methodology

[3], model [4] or framework [5]. Also, there are papers presenting development story of some tools, like [6].

Some papers describe both requirements model and its application in a specific tool. For example [7] designs a tool for management of requirements in form of specific models or [8] that defines some details about a feature management tool for product lines. Another paper [9] defines requirements as constraints and examine core concepts related to its implementation in a real tool.

There are many commercial requirements management tools, whose developers do not publish academic papers with architecture and implementation details. There are mostly marketing papers available for them. There only a few open source tools are known and cited in publications like ProR [10] or ReqLine [11].

None of the papers on the tools discusses its core model in a formal way. Some approaches and models are listed in [1] but it specifies mostly methodological aspects.

III. BASE MODEL

A. Preliminaries

The process of software development can be made in different ways. There are some general views on requirements management tool's functions but the set of requirements for this tool in specific areas may be different.

One of the ways to deal with such problem is to develop a model for that tool. This approach can be found in [7] or [8]. The model helps to define core concepts of the tool and prove some theorems over its functions.

We need to provide some terminology before starting a model. First, we will define what the *requirement* is. In this paper, *requirement* means a limitation or definition of some system's or component's functional. For our model *requirements* are unique objects that may have a specific description written by natural language and are placed in some tree structure defined in III.

B. Base model

Definition 1. A tree G is a triple (V, E, r_0) , where:

- V - a set of vertices.
- $E \subset V \times V$ - a set of edges that is an asymmetrical relation on V .

- $r_0 \in V$ - a root of the tree.
- There are no incoming edges for r_0 and there are no more than one incoming edge for the other vertices.
- All vertices are reachable from r_0 .

If $(v_1, v_2) \in E$ then v_1 is denoted as a $\text{parent}(v_2)$, while v_2 is called a child of v_1 . We define relation $\text{reachable}_E(v_1, v_2)$ as a transitive and reflexive closure of the relation E .

Definition 2. *Attributed tree* $AT = (G, \text{Key}, \text{Value}, \text{attrs})$ consists of:

- a tree $G = (V, E, r_0)$;
- a set of attribute keys Key ;
- a set of attribute values Value ;
- a functional relation $\text{attrs}: V \rightarrow (\text{Key} \rightarrow \text{Value})$ that provides each vertex with a set of attributes.

A set of all possible attributed trees is denoted as ATrees .

An attributed tree is a convenient framework to represent requirements [12] with the following semantics. If a vertex $v \in V$ represents a requirement for a target system and there are children v_1, \dots, v_n of v , then the children represent a decomposition of the requirement v . In other words, if a system satisfies to requirement v then it satisfies to all requirements v_1, \dots, v_n and vice versa.

Attributes of vertices contain various information about the requirements, for example a unique identifier, description of the requirements in natural language, its representation in a formal notation, version, etc.

An interesting particular case is the attributes, whose value is a vertex $v \in V$ or a set of vertices $vs \subseteq V$. It allows to define and to manage relations between different vertices. For example, such attributes can be used to represent traceability links between high level and low level requirements. Formally, this case is achieved if $V \cup \wp(V) \subseteq \text{Value}$.

IV. DECLARATIVE MODEL

A. The extension of the base model

The base model of requirements catalogue is an attributed tree, where each requirement has a particular set of attributes. This model is convenient for analysis of the catalogue, e.g. for formal analysis, analysis of test coverage, traceability analysis, etc. At the same time, it is difficult to manage such model manually because there are usually many interdependencies between elements and its attributes. Here and after term vertex (element of set V) and elements of requirements catalogue are used interchangeably.

That is why we introduce a declarative model of requirements catalogue that allows us to automate the handling of such dependencies. The purpose of the declarative model is to store requirements catalogue in more compact and manageable way.

The declarative model is defined stepwise. Each step is accompanied by definition of the transformation of the declarative model to the raw basic model.

B. Predicates

If requirements are developed for a product line, there is a number of requirements shared between different variations of the product. A natural wish is to have a single requirements

catalogue for the product line and the ability to build a specific one for a particular version of the product. That means there is a need to delete a subset of requirements from the catalogue if the subset is not applicable to the target product.

The similar situation happens when a catalogue is used to represent requirements of several revisions of a standard or to represent requirements of a standard with optional elements.

To introduce such ability we propose to choose especial key $\text{predicate} \in \text{Key}$, whose values are boolean. If an element has attribute predicate with value false, this element and all its children are removed from the catalogue during transformation.

The first declarative model DM_1 is an attributed tree $((V, E, r_0), \text{Key} \sqcup \{\text{predicate}\}, \text{Value}, \text{pattrs})$ that is transformed to the base model $((V', E', r_0), \text{Key}, \text{Value}, \text{attrs})$ according the following rules:

- $V' = \{v \in V: \forall v' \in V \text{ reachable}_E(v', v) \text{ predicate} \notin \text{pattrs}(v') \vee \text{pattrs}(v')(\text{predicate}) \neq \text{false}\};$
- $E' = E \cap (V' \times V');$
- $\forall v \in V' \text{ attrs}(v) = \{(k, \text{val}) \in \text{pattrs}(v): k \neq \text{predicate}\}$

C. Calculated attributes

It is an often situation when attribute value depends on values of the other attributes of the same element or even on attributes of the other elements. To express such dependencies explicitly we propose the second declarative model DM_2 that is an attributed tree $(G, \text{Key}, \text{FValue}, \text{fattrs})$, where

- $\text{FValue} = \text{Func} \times \text{Value}$;
- $\text{Func} = \text{ATrees} \times V \times \text{Key} \times \text{Value} \rightarrow \text{Value}$.

The declarative model DM_1 corresponding to the model DM_2 is an attributed tree $(G, \text{Key}, \text{Value}, \text{attrs})$:

$$\forall v \in V (k, \text{val}) \in \text{attrs}(v) \text{ iff}$$

$$\exists (k, (\text{func}, \text{fval})) \in \text{fattrs}(v): \text{val} = \text{func}(\text{AT}, v, k, \text{fval})$$

To build such requirements model it is required to solve a set of equations defined by fattrs . A simple approach is to apply fixed point iteration, while some additional implementation details will be considered in section V. There are declarative models that define a set of equations with no solutions or with non-unique solutions. A simple but reasonable limitation that allows avoiding such models is a prohibition of cyclic dependencies between attributes.

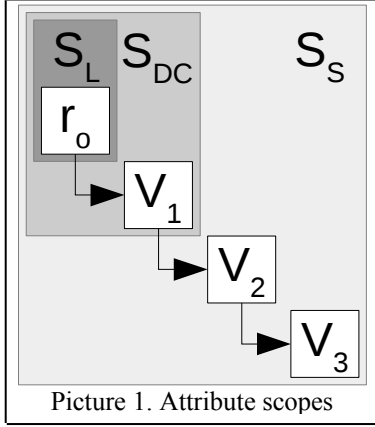
A particular case when an attribute has a constant value val is represented in the declarative model DM_2 as a pair $(\text{prj}_4, \text{val})$, where prj_4 is a projection function by the fourth argument: $\text{prj}_4(\text{AT}, v, k, \text{val}) = \text{val}$. Please note that in DM_2 predicate is considered as a regular element of the set Key .

D. Attribute scope

Another often situation happens when an attribute is applicable to the whole subtree and it has the same value for all elements. Or a similar case is when an attribute is applicable to all children of the particular element.

To handle such situations we propose the third declarative model DM_3 that is an attributed tree $(G, \text{Key}, \text{SValue}, \text{sattrs})$, where $\text{SValue} = \text{FValue} \times \text{Scope}$, $\text{Scope} = \{S_L, S_{DC}, S_S\}$ with an element having the following semantics:

- S_L – an attribute is available only in the element where it is defined.
- S_{DC} – an attribute is available in the element where it is defined and in all its direct children.
- S_S – an attribute is available in the element where it is defined and in all its successors.



An example of attribute scope can be seen in Picture 1. White rectangles are Vs. Arrows mean child-parent relation. Attribute with some scope is defined in r_0 . Grey rectangles represent different possible scopes of A and the subtrees where it will be accessible.

A transformation of declarative model DM_3 to the model DM_2 is straightforward: DM_2 is an attributed tree $(G, Key, FValue, fattrs)$, where $fattr(v) = \{k \rightarrow fval\}$ such that

- (1) $\{k \rightarrow (fval, anyscope)\} \in fattrs(v)$
- (2) $\{k \rightarrow (fval, S_{DC})\} \in fattrs(parent(v))$ if rule (1) is not applicable,
- (3) $\{k \rightarrow (fval, S_S)\} \in fattrs(v')$ if rules (1) and (2) are not applicable $\wedge reachable_E(v', v) \wedge \forall v'' \in V \forall val \in Value (reachable_E(v'', v) \wedge reachable_E(v', v'')) \Rightarrow \{k \rightarrow (val, S_S)\} \notin fattrs(v')$.

It is interesting to note that nonconstant scoped attributes can get different values in different elements because its function can depend on the vertex as a third argument.

E. Reuse of subtrees

The next item to consider is a situation when there are several subtrees of requirements that are very similar each other up to some limited number of details. In this case, it would be ideal to have a single copy of the subtree and the ability to clone it with some modifications. This approach is usually called reuse [13].

The fourth declarative model DM_4 is an attributed tree $((V, E, r_0), Key \sqcup \{cp\}, SValue, cpattr)$ with especial key cp that satisfies the following constraints:

- $\forall v \in V \quad \forall value \in Value \quad \forall s \in Scope$
 $cp \in cpattr(v) \wedge cpattr(v)(cp) = ((prj_4, val), s) \Rightarrow$
 $val \in V \wedge \forall v' \in V (v, v') \notin E;$
- $E \cup \{(v, cp(v)) \mid v \in CC(DM_4)\}$ does not contain loops, where $CC(DM_4) = \{v \in V \mid cp \in cpattr(v)\}$ and $cp(v) - val \in V$ from the constraint above.

The transformation of the model DM_4 to the model $DM_3 = ((V', E', r_0), Key, SValue, sattr)$ is performed by the following algorithm:

1. $curDM_4 := DM_4$
2. If $CC(curDM_4)$ is empty, take $DM_3 = curDM_4$ with removing cp from the Key set and finish.
3. Let $curDM_4$ is $((V, E, r_0), Key \sqcup \{cp\}, SValue, cpattr)$.

4. Choose any $v_0 \in CC(curDM_4)$ such that $\nexists v \in CC(curDM_4) reachable_E(cp(v_0), v)$. Existence of such element follows from lack of loops in $E \cup \{(v, cp(v)) \mid v \in CC(DM_4)\}$.
5. Assume without loss of generality $\forall v \in V (v_0, v) \notin V$.
6. Build $newDM_4 = ((V', E', r_0), Key \sqcup \{cp\}, SValue, cpattr')$, where
 - $V' = V \cup \{(v_0, v') \mid v' \in V \wedge reachable_E(cp(v_0), v')\}$
 - $E' = E \cup \{(v_0, (v_0, cp(v_0)))\} \cup \{(v_0, v'), (v_0, v'')\} \mid (v', v'') \in E \wedge reachable_E(cp(v_0), v')\}$
 - $\forall v \in V \setminus \{v_0\} \quad cpattr'(v) = cpattr(v)$
 - $cpattr'(v_0) = \{(k, val) \in cpattr(v_0) : k \neq cp\}$
 - $cpattr'((v_0, v')) = cpattr(v')$

Please note that $newDM_4$ satisfies both constraints of the fourth declarative model.

7. $curDM_4 := newDM_4$ and goto step 2.

Lemma 1. The algorithm terminates for any DM_4 satisfying the constraints.

The proof is based on the fact that the cardinality of $CC(curDM_4)$ is decreased every iteration because of the choice of the v_0 at step 4 that guarantees that elements with attribute cp are not cloned, while one such element loses that attribute.

Lemma 2. The result of transformation does not depend on the order of the selection of elements at step 4.

The idea of the proof is that transformations that can be chosen in non-deterministic order make modifications in non-intersecting subtrees.

Interesting to note that combination of reuse and predicate transformation can be used to define a generic subtree that is instantiated several times with different arguments using reuse transformation and the original generic subtree is eliminated with predicate transformation. Also, predicate transformation can be useful to eliminate unneeded elements from the cloned subtrees.

V. IMPLEMENTATION DETAILS

A. Identification

One of the important aspects of requirements management is requirements identification. One of the common approaches it to assign a unique identifier to each object, for example, some number or string.

In addition to that it is possible to provide each element with a qualified identifier QID defined recursively on top of identifiers ID that are unique within children of the same parent: r_0 has QID = '/ID', child v has QID = 'QID(parent)/ID'.

Let us take some example of requirements for some system. If we use QID we can have a human-readable path for each requirement. For example, we may have an element with QID = "Functional requirements/Ports/req001". As seen from the path it has a parent "Functional requirements/Ports/" and its ID is req001.

B. Calculated attributes

There are two objects related to attributes in the implementation. The first one, *attribute definition* A_DEF represents a pair (func,fval) from the formal declarative mode, where func is of type $ATrees \times V \times Key \times Value \rightarrow Value$. The second one, *attribute* A , represents a value of the attribute in the base model. A_DEF is used to calculate an actual value A when it is required.

There are several kinds of functions supported in attribute definitions.

The first kind is the *constant* functions prj_4 that always returns fval value stored in attribute definitions.

The second kind is *template functions* that stores in fval value a string with parameters encoded in curly brackets, e.g. "Hello, {K}". The value of the parameters to be used for substitution is taken from attribute with the encoded name, 'K' in the example above, of the same element.

The third kind is *formula value generator* that stores in fval value a string with an expression in a subset of JavaScript language that has access to attributes of the same element.

The fourth kind is *virtual attributes* that are implemented in Java. They have no stored fval value at all, but they have access to the whole context of the element including the complete attributed tree.

For example, Label attribute can take value of user-defined Name attribute if there is one or return system-defined identifier otherwise. Another example could be QID that calculates qualified identifier of the element as a concatenation via '/' of parent's QID with a Name of the target element.

An important additional information that the tool is able to extract from attribute definition is a set of attribute keys which values are required to calculate the actual value for the given attribute by the corresponding function.

C. Attributes life-cycle

For each attribute stored data includes function kind and fval. The pair (funckind,fval) is denoted A_ST . System-defined virtual attributes have no stored data, they are added to elements on the fly.

Let us describe a common process of attributes loading for some requirement.

1. Set of A_ST is loaded from storage to A_DEFS .
2. Set of scoped attributes that are applicable to the target one is taken from its parent and is added to A_DEFS .
3. The A_DEFS set is handled by Attribute_Calculation procedure described below.

If attributes are changed by the user using GUI session, the tool has the same A_DEFS set that contains a subset of changed attribute definitions. Then the tool applies the same Attribute_Calculation procedure as follows.

1. A_DEFS set is extended with attributes of the target element that depends on any attribute already belonging to A_DEFS .
2. The order of evaluation of attributes from A_DEFS is calculated. The order can be defined as $ORDER = (K_1..K_n)$ where K_i is the key of the attribute.
 $\forall K_i, K_j \in ORDER$ if K_j depends on K_i then $i < j$.

The algorithm is described in the next section.

3. For each A_DEF in the A_DEFS value of A is calculated and placed to AS .

After this procedure AS contains an actual state of attributes after provided changes.

D. Order extraction algorithm

As an input of order extraction, we have $KEYS = (K_1..K_n)$ that is set of attributes name in some random order and $DEPS = (K_i \rightarrow (K_{j1}..K_{jm}))$ - a map of attributes dependencies. The algorithm is as follows:

1. $ORDER$ is set to empty collection.
2. $OSET$ is the set of handling nodes.
3. Extract revert dependencies $DEPS_R$. $DEPS_K = (K_j \rightarrow (K_{i1}..K_{il}))$. If K_i depends on K_j then $DEPS_K$ contains $K_i \rightarrow K_j$ record.
4. Place $KEYS$ to $OSET$.
5. Set flag MOD is to False.
6. In $OSET$ look for candidate KK with $DEPS_K[KK] = KSET$ that complies one of following rules:
 - $KSET$ is empty.
 - OR
 - $\exists K_i \in KSET: K_i \in OSET$.
7. If K_K was found then:
 1. MOD set to True.
 2. K_K removed from $OSET$.
 3. K_K added to $ORDER$.
8. If $MOD = True \ \& \ |SET| \neq 0$ then go to step 4.

At the end of execution, the $ORDER$ will contain the order in which A 's values calculation.

E. Attribute change management

The introduction of scope and calculated attributes requires the management of attributes changes to keep all dependent attributes up-to-date.

There are two possible strategies to deal with attribute updates. The first approach is to commit all changes at runtime. The second one is to collect changes in AS and then apply them all by request. Immediate commit is tending to be simpler but more computing - intensive. Late updates require fewer calculations but need more memory. For our tool, we use the second approach because we have large catalogues with a possibility of complex relationships between its elements.

Late changes can be defined in form of new object — changes set $CS = (K \rightarrow OP, K \rightarrow A_{OLD}, K \rightarrow A_{NEW})$ where K is the key of attribute, $OP \in (remove, create, modify)$ is the operation over attribute, A_{OLD} is the value of attribute in AS before operation, A_{NEW} is the new attribute value after operation.

For attribute changes change set needs to store A_DEFS , so minimal $CS = (K, K \rightarrow A_{OLD}, K \rightarrow A_{NEW})$. To use these changes set we need to extend the model of attributes set of A .

When all attribute modifications are collected we need to apply all that changes to calculate actual values of attributes. It

is implemented in the same way as it was described in section V.C.

One more problem with attribute changes is that some of the changes need to be propagated from one requirement to another. To deal with this problem we define a concept of *change propagators*. If A_DEF (virtual attributes only for now) depends on attributes from the external element it registers a function-change propagator that is called when some change set is applied to attributes of that element. The change propagator evaluates if the changes impact the target attribute and initiate its recalculation if it is required.

F. Lazy loading

When we speak about a model of requirements in some common application like avionics we need to take into account the number of distinct requirement. Sometimes the number of artefacts for such models tends to be in the thousands or tens thousands. In that case, direct management of requirements may require a lot of resources.

To solve this problem we use the lazy loading principle. That means that AT will contain only those Vs that are requested during the usage of the model. In most cases that means that in G we have a subtree $G_L \in G$ that contains r_0 and some subtrees that are used during the current working session.

But laziness of model leads to some difficulties. First of all, we need to overlook AT instead of AT_L if we need to assure that V with given ID exists. This problem can be solved by caching id-related information in CacheStore that is always available.

G. Attribute types

In practice, the value of an attribute may have one more property – a *type*. One possible set of types includes Integer, Boolean, String, Float. Also, we may define types for Collection and Enumeration. In most cases, the value still is the simple constant.

But some attribute types cannot be defined as a single value and need to store and manage some additional data. For example, *Collection* type may use specific object $LIST = (T_v, V_1..V_n)$ where T_v is the type of collection's value and $(V_1..V_n)$ are the values stored in the collection.

One more specific type is *Enumeration*. First, enumeration requires definitions of its values. It can be made by means of $ENUM_DEF = (V_t, V_1..V_n)$ that is similar to LIST one. But to define an attribute with one selected enumeration value we need to define one more object $ENUM = (K_B, V_s)$ where K_B is the key of A with $ENUM_DEF$ and V_s is the selected value. But in a case we introduce an ENUM, we need to ensure that for every ENUM we will have an A_d where $T = ENUM_DEF$ and V_d will contain V_s .

H. References

One more problem is the implementation of relations between elements of the catalogue. Some tools manage them as the set of specific objects placed in the distinct set.

In our model relations are presented in form of specific attribute type REFERENCE. For this type we introduce value object REF_VALUE (REF, V, ERR) where REF is a string that can be resolved to V, usually containing some kind of identifier, V is the corresponding element if there is any matched by identifier, ERR is a string with an error message if REF cannot be resolved or contains incorrect value.

In this case, REF_VALUE initially contains only REF field. If someone requires the result of REF_VALUE resolution then the tool tries to resolve the REF and then fills V or ERR.

References are also required some additional handling to support its consistency. In a case REF or target V is changed we may need to track its changes and update related REF_VALUE.

One more specific problem is reverted links. If we have a relation $V_1 \rightarrow V_2$ we may need to know for V_2 that it has a relation to V_1 . This kind of relations is called "reverse references".

If links are stored in AT then we may use one more function $(V_2, LN) \rightarrow V_1$ to store reverse relations. If we define a new type of attributes or the specific state of REF_VALUE then we face a problem of keeping it up-to-date.

In our model, we store reverts links in the cache in form of $(V_2, LN) \rightarrow (V_1..V_n)$ function. That allows us to easily get revert links on V_2 if the state of cache is valid.

In a case of completely loaded AT the problem is not so difficult to solve because we always have the actual state of every V. But we cannot guaranty the V's state in case of a partially loaded AT that happens in case of lazy loading.

If we have some loaded $AT_L \subseteq AT$, relation $(V_1, LN) \rightarrow V_2$, $V_1 \notin AT \wedge V_2 \in AT$ then if we need to get revert links on V_2 we may need to load the whole AT to be sure that all possible V_1 were found.

In our case, this problem is solved by storing reverse links in the cache. But in this case, we still have one necessary problem. Let us introduce some link $L(V_1, V_2, LN)$. If we already resolve this link then the record in cache tends to be present. But what if we introduce V_2 in the model when V_1 is loaded and the link is resolved was not found? The situation takes place when V_2 is loaded by the lazy method, created or modified.

In the worst case, we need to track changes of the whole AT for all links. A better solution is to manage some kind of scope for which link tends to be resolved. That is not implemented yet, but it is in our plans.

Relations can be used for some specific activities. One of them is changes management. Changes management is performed when some V_1 with links $(L_1..L_n)$ is changed. In this case, some operations will be performed on V's obtained from $L_1..L_n$. The nature of such operation can be different. For some tools, those Vs will be marked in a model with the specific flag. In other cases, the models can define additional actions depending on the kind of change.

CONCLUSION

We presented a formal metamodel that is used as a basis for building Requality requirements management tool. We covered different difficulties related to its implementation. But the experience demonstrates that the model allows handling quite big requirements catalogue with many relations between its elements.

The future work includes analysis and implementation of new kinds of functions for calculated values and development of user-friendly patterns for solving common user tasks on top of the semantics defined in the paper.

REFERENCES

- [1] S. Hallerstede, M. Jastram, L. Ladenberger "A method and tool for tracing requirements into specifications" *Science of Computer Programming* 82, 2014, pp. 2–21.
- [2] R. Thayer. M. Dorfman "Software Engineering" IEEE Computer Society press, 1997.
- [3] M. Palumbo "Requirements Management for Safety Critical Systems" unpublished.
- [4] P. Roques "How modeling can be useful to better define and trace requirements" *The Magazine for RE Professionals from IREB* Issue 2015-02, 2015.
- [5] Open Group Standard "Dependability through Assuredness™ (O-DA) Framework", The Open Group Releases, 2013.
- [6] A. Nordin, A. Ikhwan Omar, M. Usamah Megat Mohamed Amin, N. Salleh "Development of scenario management and requirements tool (SMaRT): towards supporting scenario-based requirements engineering methodology" *International Journal of Engineering & Technology* 7, 2017, pp 62-65.
- [7] D. Lozhkina, S. Staroletov "An online tool for requirements engineering, modeling and verification of distributed software based on the MDD approach" *Proceedings of the 11th Spring /Summer Young Researchers' Colloquium on Software Engineering*, 2017, pp. 23-28.
- [8] T. von der Maßen, H. Lichter "RequiLine: A Requirements Engineering Tool for Software Product Lines", *Software Product-Family Engineering*, 2003, Heidelberg, pp. 168-180.
- [9] N. W. Mogk "A Requirements Management System based on an Optimization Model of the Design Process", *Conference on Systems Engineering Research (CSER 2014)*, 2014, pp 21-22
- [10] "ProR Requirement Engineering Platform". [Online]. <http://www.eclipse.org/rmf/pror/>. [Accessed: 2-Apr-2018].
- [11] "ReqLine Download (ReqLine.exe)." [Online]. Available: <http://downloads.informer.com/reqline/>. [Accessed: 3-Apr-2018].
- [12] Alexey Khoroshilov "On formalization of operating systems behaviour verification". // In *Proceedings of 11th International Conference on Computer Science and Information Technologies (CSIT-2017)*, pp. 168-172, September 25 - 29, 2017, Yerevan, Armenia. DOI: 10.1109/CSITechnol.2017.8312164.
- [13] W. Frakes, C. Terry "Software Reuse: Metrics and Models". *ACM Computing Surveys* Vol. 28, No. 2, 1996.

Extracting architectural information from source code of ARINC 653-compatible application software using CEGAR-based approach

Sergey Lesovoy

Ivannikov Institute for System Programming of the RAS
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation
lesovoy@ispras.ru

Abstract. The paper describes an algorithm of extracting architectural information from source code of ARINC 653-compatible application software. The extracted architectural information can be further used for creation the architecture models of the system. Architecture models are used in model-based development process for creation of the complex systems such as Integrated Modular Avionics (IMA) systems. For extracting architectural information from source code CEGAR-based approach is used. Counterexample-guided abstraction refinement (CEGAR) is a technique widely used in software model checking. The algorithm is implemented in CPAchecker tool.

Keywords: architectural information, architecture models, ARINC 653, IMA, CEGAR.

I. INTRODUCTION

The purpose of using architecture models is to analyze the system's static and dynamic features during the development process. These features may include real-time performance, resources consumption, reliability etc. This aspect is extremely important while developing complex systems that include both software and hardware components produced by the different suppliers. Using model-based approach at the early stages of the development of the project will help to avoid a waste of time and money for correction of system defects when the system is created. For creation of the architectural models various modeling languages can be used. The most popular ones used for architecture modelling are SysML[1] and AADL[2,3].

The model-based development process includes two major project steps. At the first step the system model is being created. There are different levels for representation of the system model. The primary focus of this paper is the architectural models. On the second step of the project the system model will be used as input for detailed design and system implementation. This step may also include the model transformations to some intermediate formats used in system design and implementation. In the ideal case the system model can be transformed to the source code of the system.

It may be useful to analyze and reuse some components of legacy systems during development of new systems. By using a model-based approach it is possible to build an architecture model from the existing source code of the legacy system. This model can be used as for system analysis as well as for reusing some components of the legacy system in the new design. In many cases it will allow to avoid creation of a new system from scratch.

A process of model creation for existing system is called a model-driven reverse engineering (MDRE). If the source code of a legacy system is available then it is possible to build a system model from its source code. This process contains two steps. The first step is source code analysis. The second step is model transformations to the target output format. This paper describes the first step – source code analysis for application software that is based on Integrated Modular Avionics (IMA) architecture and ARINC 653 specification. The goal of source code analysis is to extract architecture information that is necessary for creation of the architecture model of the system.

The rest of the paper is organized as follows. Section 2 provides an overview of IMA architecture and ARINC 653 specification. It also contains a simple example of source code to be used for further analysis. Section 3 describes the concept of architectural information in source code, a general approach and a particular algorithm used for extracting architectural information from source code. Section 4 describes the results and outlines the future research and development tasks.

II. INTEGRATED MODULAR AVIONICS AND ARINC 653

IMA architecture is widely used in avionics industry for implementation the safety critical applications. In IMA systems multiple avionics applications can share resources of a single hardware platform (core module) without any mutual influence. ARINC 653 [4] is a set of documents that define the requirements for software components of IMA systems. The key concept of ARINC 653 is a partition. ARINC 653 compatible Operating System (OS) provides a dedicated portion of memory and predefined time slot within a fixed schedule for each partition. It prevents any affect from software executing in one partition to software in other partitions.

ARINC 653 specification defines that IMA system may include the following software components: core software, application partitions and system partitions. Core software consists of OS and Application/Executive (APEX) interface. The APEX interface defines a set of services provided by the OS for application software. In each application partition can be allocated only one application. System partitions contain system software that can directly interact with the OS without using APEX interface.

Communications between applications allocated in different partitions is called the interpartition communication. The interpartition communication is only available via communication channels. To access a communication channel the application can use the ports created inside a partition. ARINC 653 supports two port types: sampling ports and queuing ports.

An application software compliant with ARINC 653 specification has a typical structure. Such a software can be located in a single partition or in multiple partitions. To access the various services of ARINC 653 based OS an application software uses function calls defined in the APEX interface. For each partition several processes can be created. One process is responsible for partition initialization. This process creates other processes and various objects. Finally, when the initialization of partition has been finished this process sets the partition to NORMAL state using SET_PARTITION_MODE function call. Since this moment a scheduling for all processes created inside a partition is started. It is important to note that after the initialization of partition has been finished there is no way to create any new processes and objects.

An ARINC 653 process is quite similar to a POSIX thread. To create a process, it is necessary to create a structure that contains the process's attributes and pass it to CREATE_PROCESS function. ENTRY_POINT is an attribute of the process that contains the address of the function that will be called when the process is started. This function implements the application logic of the process and its communication procedures with other processes.

Communications between processes within a single partition is called intrapartition communication. Buffers and blackboards are used for communication between processes inside a partition. Semaphores, events and mutexes are used for process synchronization. Any objects for communication and synchronization can be created using function calls defined in the APEX interface. The processes located inside the same partition can also communicate via global variables.

At the end of this section a simple example of application software will be demonstrated and explained. The source code fragment of the application software compliant with ARINC 653 specification is shown in Fig.1. Source code fragment in Fig.1 includes three functions: Run_10_Hz, Run_Monitor and main. In function main two processes, one event and three sampling ports are created.

```
static void Run_10_Hz(void) {
    ...
    while (1) {
        SET_EVENT ( wakeup, ret );
        READ_SAMPLING_MESSAGE(port_raw_data,
            (MESSAGE_ADDR_TYPE)&sensor_data,
            &len, &validity, &ret);
        // Some operations with data ...
        WRITE_SAMPLING_MESSAGE(port_data_out,
            (MESSAGE_ADDR_TYPE)&output_data,
            &len2, &ret);
        PERIODIC_WAIT(&ret_pause); }
    }

static void Run_Monitor(void) {
    while (1) {
        WAIT_EVENT ( wakeup, TimeOut, ret );
        RESET_EVENT ( wakeup, ret );
        // Some operations with data ...
        WRITE_SAMPLING_MESSAGE( port_status,
            (MESSAGE_ADDR_TYPE)&status_data,
            &len, &ret ); }
    }

void main(void) {
    PROCESS_ATTRIBUTE_TYPE Proc_10_Hz_Attributes;
    Proc_10_Hz_Attributes.ENTRY_POINT = Run_10_Hz;
    Proc_10_Hz_Attributes.PERIOD = 100000000LL;
    strncpy(Proc_10_Hz_Attributes.NAME, "Proc_10_Hz",
        sizeof(PROCESS_NAME_TYPE));
    CREATE_PROCESS( &Proc_10_Hz_Attributes, &pid_p0,
        &ret );
    START( pid_p0, &ret );

    PROCESS_ATTRIBUTE_TYPE Proc_Monitor_Attributes;

    Proc_Monitor_Attributes.ENTRY_POINT = Run_Monitor;

    Proc_Monitor_Attributes.PERIOD =
        INFINITE_TIME_VALUE;
    strncpy(Proc_Monitor_Attributes.NAME, "Proc_Monitor",
        sizeof(PROCESS_NAME_TYPE));
    CREATE_PROCESS( &Proc_Monitor_Attributes, &pid_p1,
        &ret );
    START( pid_p1, &ret );

    EVENT_NAME_TYPE EventName;
    strncpy( EventName, "Wakeup", ...);
    CREATE_EVENT ( EventName, wakeup, ret );
    ...
    CREATE_SAMPLING_PORT( "RAW_DATA",
        port_size, DESTINATION, period, &port_raw_data, ...);
    CREATE_SAMPLING_PORT( "DATA_OUT",
        port_size, SOURCE, period, &port_data_out, ...);
    CREATE_SAMPLING_PORT( "STATUS", port_size,
        SOURCE, period, &port_status ...);

    SET_PARTITION_MODE ( NORMAL, &ReturnCode );
    return 0;
}
```

Fig. 1. Source code fragment with APEX calls.

For process creation, APEX call `CREATE_PROCESS` is used. The first argument of `CREATE_PROCESS` has a type `PROCESS_ATTRIBUTE_TYPE`. It is a structure that contains attributes for the created process. The `ENTRY_POINT` attribute is equal to `Run_10_Hz` for the first process and is equal to `Run_Monitor` for the second one. `Run_10_Hz` and `Run_Monitor` are the function's names that are called when the processes are started.

Below in the main function, APEX call `CREATE_EVENT` is used to create an event object. An event object has a name `Wakeup`. Then APEX calls `CREATE_SAMPLING_PORT` are used to create three sampling ports. These ports have the following names: `RAW_DATA`, `DATA_OUT` and `STATUS`. In the end of the main function APEX call `SET_PARTITION_MODE` is used to set the partition to the `NORMAL` state. After that, OS will invoke functions `Run_10_Hz` and `Run_Monitor`.

A function `Run_10_Hz` is called periodically with period 10 milliseconds. This value for period was set in `PERIOD` attribute during the creation of the first process. Each time when the function `Run_10_Hz` is called, it activates the event `Wakeup`, reads a message from sampling port `RAW_DATA`, performs some operations with data and writes a message to sampling port `DATA_OUT`.

Function `Run_Monitor` belongs to the second process that is an aperiodic. This function waits for event `Wakeup`, resets it, performs some operations with data and writes a message to sampling port `STATUS`.

III. SOURCE CODE ANALYSIS

A. Architectural information in source code

The main goal of source code analysis in the paper is to extract the architectural information from it. Architectural information in source code of application software compliant with ARINC 653 specification includes the processes in each partition and their attributes, all objects created for interpartition and intrapartition communications and their attributes. It also includes the ways of communications and synchronizations between processes located inside the same partition or in different partitions. If the global variables are used for communication between processes inside partition then these variables also should be considered as architectural information.

The source code fragment in Fig.1 contains the following architectural information: two processes, one event and three sampling ports. Attributes of each process and each object (event, port) are also important architectural information. For synchronization between two processes the event object is used. In the first process APEX call `SET_EVENT` is used to activate an event. The second process uses APEX call `WAIT_EVENT` for receiving this event. Sampling ports are used in both processes to communicate with external environment, i.e. with processes allocated in other partitions or with external devices.

The source code of real avionic application can contain hundreds of processes communicating with each other and with external environment via large number of the objects. Extracting such architectural information from source code can be time consuming task. This paper proposes a way to do it automatically. The next sections describe a general approach and a particular algorithm used for source code analysis.

B. General approach for source code analysis

For source code analysis an approach based on Counterexample-guided abstraction refinement (CEGAR) algorithm is used. In CPAchecker tool [5] the basic predicate-based CEGAR algorithm has been extended for explicit-value analysis [7]. CPAchecker is a tool for configurable program analysis (CPA) [5,6] that combines the traditional program analyses and software model checking. In this paper the extended for explicit-value analysis CEGAR algorithm is applied for the task of extracting architecture information from source code. The algorithm is implemented in CPAchecker tool.

The algorithm presented in this paper uses a Control-Flow Automata (CFA) as intermediate representations of the program to be analyzed. CFA is a directed graph containing nodes and edges. A node corresponds to a program location. An edge corresponds to a certain operation of the program, for example, an assignment statement, a conditional branch or a function call. During the analysis, the algorithm constructs an Abstract Reachability Graph (ARG) using a program CFA. ARG is also a directed graph but its nodes correspond to abstract states of the program. Each abstract state contains a program location, a data state and a call stack. A data state is a mapping between program variables and their values. In data state some program variables may not have the values.

ARG represents possible execution paths of the program. It means that ARG can contain both feasible (real) program paths as well as the infeasible (spurious) paths. The program path is feasible if it can be executed at runtime otherwise it is infeasible. A path in ARG is a sequence of abstract states connected by edges. An abstract state is reachable if there is a feasible program path that contains this state.

C. Extracting APEX calls from source code

Before starting the algorithm description, it is necessary to explain some important concepts used by the algorithm. The algorithm constructs the ARG by sequentially adding the new abstract states to it. For the current state the algorithm gets the list of all its successors and adds each of them to ARG. There is an edge between the current state and each its successor.

Target states.

Some edges may correspond to a function call in source code. If this function is defined in APEX interface, the algorithm will need to collect additional information about this function call.

An abstract state in ARG which immediately follows such a function call is called the **target state**. Any target state has an incoming edge with APEX call. For each target state there is a path in ARG from the initial state to it. The algorithm performs a feasibility check for these paths.

Precision.

Explicit-value analysis tracks values for the program variables. In many cases it is enough to track only a part of program variables that are important for a particular analysis. A set of program variables that are being tracked for the current abstract state is called a **precision**. Different abstract states may have different precisions. The empty precision means that no variables are being tracked. The full precision means that all variables are being tracked. As described in [6] the value analysis algorithm implemented in CPAchecker can change a precision during the analysis depending on some conditions. It is called a precision adjustment.

An edge in ARG can correspond to a program operation that changes a value of a program variable. For example, an assignment operation changes the value of the left-hand operand, for a function call the values of arguments are assigned to function's parameters, etc. When the algorithm handles an edge between the current state (predecessor) and next state (successor) it uses a precision of the predecessor. If precision of the predecessor contains the current variable, then the algorithm evaluates and stores its new value in abstract state. The algorithm of analysis can use the values of variables stored in abstract states for different purposes.

Fig. 2 presents a pseudocode of the main algorithm for extracting the architectural information from source code. This algorithm implements a classical CEGAR cycle extended for explicit value analysis [7] and is applied for the task of extracting architectural information from source code.

CFA of the program is used as an input data for the algorithm. The algorithm uses two variables to store the abstract states: "reached" and "waitlist". A variable "reached" contains the set of abstract states that have been explored already. A variable "waitlist" contains the set of abstract states that have to be explored on the next steps of the algorithm.

At the beginning the algorithm takes the initial state from CFA and put it to "waitlist". After that the external loop of the algorithm begins. The algorithm takes and removes the current state from waitlist. Further the algorithm gets all reachable successors for the current state using function "getAbstractSuccessors". A pseudocode for the function "getAbstractSuccessors" is shown on Fig.3. The first operation of the function gets all successors (CFA nodes) of the current state. Then the function consecutively handles the edges (function "handleEdge") between the current state and each its successor. The function "handleEdge" takes two parameters. The first parameter is an edge to be explored. The second parameter is a precision. The precision is taken from the edge predecessor. Depending on the operation in source code that the edge corresponds to, the function "handleEdge" performs the following actions:

- For an assignment operation, the algorithm evaluates a new value for this variable. The new value for a variable will be stored in abstract state if this variable is contained in the precision.
- For a function call the function's arguments are assigned to function's parameters.
- For a conditional branch a logical value for a condition is evaluated. If the logical value of a conditional

branch is equal to FALSE then the function "handleEdge" return FALSE. It means that this successor is not reachable. In all other cases the function returns TRUE and the current successor is added to the list of reachable successors. So, function "getAbstractSuccessors" returns for the current state a list of all its reachable successors.

FUNCTION main

INPUT

CFA of the program;

OUTPUT

Architectural information

VARIABLES

reached – a set of states that have been reached;

waitlist – a set of states to be explored;

BEGIN

initState = getInitialState(CFA);

addStateToWaitlist(initState);

// Traverse through all CFA nodes.

LOOP WHILE waitlist \neq 0 // External loop.

curState = getAndRemoveStateFromWaitlist();

// Get all reachable successors of the current state.

successors = getAbstractSuccessors(curState);

// Traverse through all reachable successors.

FOR EACH nextState **IN** successors // Internal loop.

IF isTargetState(nextState)

path = getPathToState(nextState);

IF isPathFeasible(path) = FALSE

// Refine the path.

performRefinementForPath(path,

reached, waitlist);

BREAK // Go to external loop.

END IF

END IF

merge(nextState, reached);

update(reached);

addStateToWaitlist(nextState);

END FOR EACH

END LOOP

END

Fig. 2. The main algorithm for extracting the architectural information from source code.

Further in internal loop the main algorithm traverses through all reachable successors for the current state. At this part of algorithm, a successor is called as a "nextState". The algorithm checks whether a nextState is a target state. If it is a target state, the algorithm calculates a path in ARG from the initial state to the current target state and checks its feasibility using function "isPathFeasible". In a classical CEGAR algorithm a path in a program to be explored is called a counterexample and it means a path to the error state. In the current algorithm it is just a path to the target state we need to explore.

The algorithm of function "isPathFeasible" is shown in Fig. 4. To check the path feasibility the algorithm consecutively passes through all edges of the path, starting from the initial state. The algorithm analyses the operations for

each edge. To track all program variables, for each state on the path the full precision is set, i.e. the algorithm performs the feasibility check for a path with the full precision.

```

FUNCTION getAbstractSuccessors
INPUT
    curState // Current state.
RETURN
    reachableSuccessors // All reachable successors.
BEGIN
    // Get all successors of the current state.
    allCFASuccessors = getAllSuccessors(curState);
    FOR EACH successor IN allCFASuccessors
        edge = getEdge(curState, successor);
        precision = getPrecisionForState(curState);
        IF handleEdge(edge, precision) = TRUE
            // Add successor to reachableSuccessors.
            addToSet(reachableSuccessors, successor);
        END IF
    END FOR EACH
    RETURN reachableSuccessors;
END

```

Fig. 3. The algorithm of function getAbstractSuccessors.

Each edge on the path is handled with the function “handleEdge” that was already described above. For a conditional branch the function “handleEdge” may return FALSE if logical condition is not satisfied. The path is not feasible if for any edge on the path the logical condition is not satisfied. In this case the function “isPathFeasible” returns FALSE. In all other cases the path is feasible. If the path is feasible then at the last state of the path the values for all program variables assigned on this path are known. The last edge and the last state of the path is passed to a function “handleApexCall”.

```

FUNCTION isPathFeasible
INPUT
    path
RETURN
    TRUE – path is feasible;
    FALSE – path is not feasible;
BEGIN
    // Traverse through all edges.
    FOR EACH edge IN path
        precision = FULL;
        IF handleEdge(edge, precision) = FALSE
            RETURN FALSE
        END IF
        IF isLastEdge(edge, path) = TRUE
            lastState = getSuccessor(edge);
            handleApexCall(edge, lastState);
        END IF
    END FOR EACH
    RETURN TRUE
END

```

Fig. 4. The algorithm of function isPathFeasible.

The last edge contains the information about the APEX call. The last state contains values for all program variables on the path. The function “handleApexCall” extracts all architectural information including the function name for the last APEX call, values for its argument and call stack. It is important to note that the algorithm extracts architecture information only from the APEX calls that belong to a feasible paths. The algorithm collects the architectural information for each APEX call and uses it as output data. The format of the output data will be described in the next section.

If the algorithm has detected that a path is infeasible, then it will refine this path. During refinement procedure the precision for some abstract states of the path are changed by adding variables for tracking. The refinement procedure is described in detail in [7]. Finally, the algorithm will update the ARG in such a way that will eliminate the infeasible path or its part for the further analysis.

At the end of the internal loop the algorithm tries to merge the nextState with already reached states, updates reached states and adds the last explored state (nextState) to “waitlist”. These steps are described in details in [7] (see section “Reachability Algorithm for CPA”).

The described above steps of internal loop are being repeated for each reachable successor of the current state.

Then the algorithm leaves the internal loop and continues its execution by taking the first step on the main loop. It takes the next state from “waitlist” variable and repeats all steps already described above. The algorithm terminates when all ARG abstract states have been processed.

D. Output format

The algorithm keeps the collected architectural information in the internal format. For further processing the architectural information has to be transformed to the external representation. The export format depends on the tool that is used for creation the architecture models. The architectural information can also be exported to human-readable format. In Fig.5 the architectural information extracted by the algorithm from the source code fragment in Fig.1 is presented in a human-readable format. The presented architectural information is divided onto sections. The first section contains information about ARINC653 processes. There are two processes with names Proc_10_Hz and Proc_Monitor. Below the process name there are the list of its attributes. On the Fig3 there are only three attributes are presented: PROCESS_ID, ENTRY_POINT and PERIOD. PROCESS_ID is a serial number of the process inside a partition. ENTRY_POINT is a name of the function that is being called when the process is started. PERIOD shows the period’s duration in milliseconds. INFINITE_TIME_VALUE in source code corresponds to aperiodic process. The next sections contain the information about other ARINC653 objects created in the source code.

ARINC653_SAMPLING_PORTS section shows three sampling ports and its attributes.

ARINC653_SAMPLING_MESSAGES section shows what processes are using sampling ports for sending (WRITE subsection) and for receiving (READ subsection) messages.

For example the port DATA_OUT is used by the first process (function Run_10_Hz) for sending messages.

ARINC653_EVENTS contains information about the events that have been created and used in the source code.

ARINC653_EVENTS section has three subsections: SET_EVENT, WAIT_EVENT and RESET_EVENTS. The name of the subsection corresponds to the APEX call. For example, a subsection SET_EVENT corresponds to APEX call SET_EVENT that activate an event.

```

==ARINC653_PROCESSES==
Proc_10_Hz
  PROCESS_ID: 0
  ENTRY_POINT: Run_10_Hz(0)
  PERIOD = 100 ms
...
Proc_Monitor
  PROCESS_ID: 1
  ENTRY_POINT: Run_Monitor(1)
  APERIODIC
...
==ARINC653_SAMPLING_PORTS==
1) RAW_DATA
  MAX_MESSAGE_SIZE = 128
  PORT_DIRECTION = DESTINATION
  REFRESH_PERIOD = 1000
...
2) DATA_OUT
...
3) STATUS
...
==ARINC653_SAMPLING_MESSAGES==
=WRITE=
1) PORT_NAME=DATA_OUT;
  ENTRY_POINT=Run_10_Hz(0);
2) PORT_NAME=STATUS;
  ENTRY_POINT=Run_Monitor(1);
=READ=
1) PORT_NAME=RAW_DATA;
  ENTRY_POINT=Run_10_Hz(0);
...
==ARINC653_EVENTS==
=SET_EVENT=
1) EVENT_NAME=Wakeup;
  ENTRY_POINT=Run_10_Hz(0)
=WAIT_EVENT=
1) EVENT_NAME=Wakeup;
  ENTRY_POINT=Run_Monitor(1)
=RESET_EVENTS= ...

```

Fig. 5. The architectural information in human-readable format.

In the analyzed source call there is only one such a call for event with a name Wakeup. The ENTRY_POINT string contains a name of the ENTRY_POINT function where this call was made. In the real code the ENTRY_POINT function is determined using a call stack information. The serial number of the process is shown in parentheses. In the Fig.3 we can see that event Wakeup was set in function Run_10_Hz that

belongs to the process with PROCESS_ID equal to 0 (Proc_10_Hz). From the section WAIT_EVENT we can understand that the function Run_Monitor waits for the event Wakeup using APEX call WAIT_EVENT. The function Run_Monitor belongs to process Proc_Monitor. So, we can see that the event Wakeup is used by two processes for synchronization.

The representation of architectural information in the human-readable format is presented only for explaining the content of such information and is useful mainly for debug purposes. As it was mentioned above for further processing the architectural information should be transformed to the format that is supported by the external tools.

IV. RESULTS AND CONCLUSIONS

The algorithm presented in the paper allows extracting architectural information from source code of ARINC 653-compatible application software. The main contribution of this paper is the application the ideas of counterexample and path feasibility check for the task of extracting the architectural information from source code. In the presented algorithm the task of extracting architectural information from source code has been solved by transforming it into the task of path feasibility check.

The work of the algorithm is demonstrated on the simple example. By this moment the algorithm has been tested on the several software applications that are compatible with ARINC 653 specification. These applications contained up to 50 ARINC 653 process and up to 30 objects for communications.

The next task to be done is to extend the algorithm for extracting from source code the global variables that are used for communication between processes inside partition. It is also necessary to implement the algorithm of transformation of the architecture information to the architecture model.

REFERENCES

- [1] OMG Systems Modeling Language (OMG SysML™) Version 1.5, 2017. [Online]. Available: <http://www.omg.org/spec/SysML/1.5/>
- [2] SAE International standard AS5506C, Architecture Analysis & Design Language (AADL), 2017. [Online]. Available: <http://standards.sae.org/as5506c/>.
- [3] Feiler P., Gluch D., Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, 2012.
- [4] ARINC Specification 653P1-4. Avionics Application Software Standard Interface Part 1 – Required Services. Published by SAE-ITC, Maryland, USA. August 21, 2015.
- [5] [Online]. Available: <https://cpachecker.sosy-lab.org/>
- [6] D. Beyer, T. A. Henzinger and G. Theoduloz, Program Analysis with Dynamic Precision Adjustment, 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, L'Aquila, 2008, pp. 29-38. doi: 10.1109/ASE.2008.13
- [7] Beyer D., Löwe S. (2013) Explicit-State Software Model Checking Based on CEGAR and Interpolation. In: Cortellessa V., Varró D. (eds) Fundamental Approaches to Software Engineering. FASE 2013. Lecture Notes in Computer Science, vol 7793, pp 146-162. Springer, Berlin, Heidelberg [Online]. Available: https://doi.org/10.1007/978-3-642-37057-1_11