

SYRCoSE 2014

Editors:

Alexander Kamkin, Alexander Petrenko and
Andrey Terekhov

Preliminary Proceedings of the 8th Spring/Summer Young Researchers'
Colloquium on Software Engineering

Saint Petersburg, May 29-31, 2014

Preliminary Proceedings of the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014), May 29-31, 2014 – Saint Petersburg, Russia:

The issue contains the papers presented at the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014) held in Saint Petersburg, Russia on May 29-31, 2014. Paper selection was based on a competitive peer review process being done by the program committee. Both regular and research-in-progress papers were considered acceptable for the colloquium.

The topics of the colloquium include formal methods, embedded system design, system programming, process mining, testing, compiler technologies and others.

Предварительный сборник трудов 8-ого весеннего/летнего коллоквиума молодых исследователей в области программной инженерии (SYRCoSE 2014), 29-31 мая 2014 г. – Санкт Петербург, Россия:

Сборник содержит статьи, представленные на 8-ом весеннем/летнем коллоквиуме молодых исследователей в области программной инженерии (SYRCoSE 2014), прошедшем в Санкт Петербурге 29-31 мая 2014 г. Отбор статей производился на основе рецензирования материалов программным комитетом. На коллоквиум допускались как полные статьи, так и краткие сообщения, описывающие текущие исследования.

Программа коллоквиума охватывает следующие темы: формальные методы, проектирование встроенных систем, системное программирование, анализ процессов, тестирование, компиляторные технологии и др.

ISBN 978-5-91474-020-4

Contents

Foreword.....	5
Committees / Referees.....	6
Formal Methods	
Modular Construction of Time Petri Nets Reachability Graph <i>I. Knizhnikova, L. Dworzanski</i>	8
On the Deadlock Control in Parallel Resource-Constrained Workflows <i>V. Bashkin, N. Panfilova</i>	13
LTL-Specification, Verification and Construction of PLC Programs <i>D. Ryabukhin, E. Kuzmin</i>	19
An Approach to Lightweight Static Data Race Detection <i>P. Andrianov, A. Khoroshilov, V. Mutilin</i>	27
Minimizing the Number of Static Verifier Traces to Reduce Time for Finding Bugs in Linux Kernel Modules <i>V. Mordan, E. Novikov</i>	34
Tools Support for Linux Kernel Deductive Verification Workflow <i>D. Efremov, N. Komarov</i>	40
Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog <i>R. Haberland, S. Ivanovskiy</i>	46
Embedded System Design and System Programming	
Energy-Aware Design of Embedded Software through Modelling and Simulation <i>J.A. Esparza Isasa, P.G. Larsen, F.O. Hansen</i>	51
Energy Aware Congestion Management in Dynamic Wireless Mesh Network <i>S.P. Shiva Prakash, T.N. Nagabhushan, K. Krinkin, O. Sholokhova</i>	57
An Architecture of Effective Discrete-Event Simulation Engine for Early Validation of Avionics Systems <i>D. Buzdalov</i>	65
Protecting Applications from Highly Privileged Malware Using Bare-metal Hypervisor <i>K. Mallachiev, N. Pakulin</i>	71
Process Mining	
Checking Conformance of High-Level Business Process Models to Event Logs <i>A. Begicheva, I. Lomazova</i>	77
Applying Graph Grammars for the Generation of Process Models and Their Logs <i>V. Kataeva, A. Kalenkova</i>	83
Generation of a Set of Event Logs with Noise <i>I. Shugurov, A. Mitsyuk</i>	88
DPMine/C: C++ Library and Graphical Frontend for DPMine Workflow Language <i>S. Shershakov</i>	96

Component-based VTMine/C Framework: Not Only Modelling <i>P. Kim, O. Bulanov, S. Shershakov</i>	102
Testing	
Extended Finite State Machine based Test Derivation Strategies for Telecommunication Protocols <i>N. Kushik, A. Kolomeez, A. Cavalli, N. Yevtushenko</i>	108
A Generic Knowledgebase for Test Generation <i>A. Kotsynyak, A. Tatarnikov</i>	114
Interactive Test Case Design via Attribute Exploration <i>F. Strok, G. Kondratiev</i>	118
Keyword-Driven Testing with Message Sequence Charts <i>B. Tyutin, A. Veselov, V. Kotlyarov</i>	122
Reconciliation Testing Aspects of Trading Systems Software Failures <i>A.-M. Kriger, V. Isayev, A. Pochukalina</i>	125
Simulation-based Hardware Verification Back-end: Diagnostics <i>M. Chupilko, A. Protsenko</i>	130
Compiler Technologies	
From Abstract Parsing to Abstract Translation <i>S. Grigoriev, Ia. Kirilenko</i>	135
Comparison of Generalized Ascent and Descent Parsers <i>A. Ragoza, S. Grigoriev</i>	140
One Approach to Automated Compiler Verification <i>V. Bessonov, L. Lyadova</i>	143
Generation of Overlapped Executable Code <i>V. Aranov, A. Terentiev</i>	150
Application-Specific Methods and Tools	
Predicative Analytics for Developing Software <i>N. Yarushkina, T. Afanasieva, I. Timina</i>	154
Detecting and Highlighting Text in Images <i>I. Pakhomov</i>	159
Using Multidimensional Ontology of Electronic Document for Solving Semantic Indexing Problem <i>V. Lanin, G. Sokolov</i>	166
Generation of Domain-Specific Languages on the Basis of Ontologies <i>A. Sukhov</i>	170
Dynamic Information Model Interactions: Design and Implementation of Database-Driven Workflow Approach <i>A. Petrov</i>	177

Foreword

Dear participants, we are glad to meet you at the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). The event is held in Saint Petersburg, the second largest city in Russia and its cultural capital. The colloquium is hosted by Saint Petersburg State Polytechnical University (SPbSPU), one of the top research and educational institutions in Russian Federation in the field of applied physics and mathematics, industrial engineering, chemical engineering, aerospace engineering and many other disciplines. SYRCoSE 2014 is organized by Institute for System Programming of the Russian Academy of Sciences (ISPRAS) and Saint Petersburg State University (SPbSU) jointly with SPbSPU.

In this year, Program Committee (consisting of 50 members from more than 25 organizations) has selected 31 papers. Each submitted paper has been reviewed independently by three referees. Participants of SYRCoSE 2014 represent well-known universities, research institutes and companies such as Aarhus University, Exactpro Systems, ISPRAS, JSS Research Foundation, JSS Academy of Technical Education, Kostroma State Technological University, National Research University – Higher School of Economics, Obninsk Institute for Nuclear Power Engineering, Perm State National Research University, Saint Petersburg Electrotechnical University “LETI”, SPbSPU, SPbSU, TELECOM SudParis, Tomsk State University, Ulyanovsk State Technical University, Yandex, Yaroslavl State University and Yuri Gagarin State Technical University of Saratov (4 countries, 13 cities and 18 organizations).

We would like to thank all of the participants of SYRCoSE 2014 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, Bertrand Meyer (ETH Zürich, Switzerland) and Kostya Serebryany (Google Moscow, Russia). We would also like to thank our sponsors and supporters: Russian Foundation for Basic Research (grant 14-07-06006), Google, Exactpro Systems and CyberLeninka. Finally, our special thanks to local organizers, Igor Chernorutskiy, Vsevolod Kotlyarov and Tatyana Elamic (SPbSPU), for their invaluable help in organizing the colloquium in Saint Petersburg.

Sincerely yours

Alexander Kamkin, Alexander Petrenko and Andrey Terekhov
May 2014

Committees

Program Committee Chairs

 Alexander Petrenko – Russia
Institute for System Programming of RAS

 Andrey Terekhov – Russia
Saint-Petersburg State University

Program Committee

 Jean-Michel Adam – France
Pierre Mendès France University

 Sergey Avdoshin – Russia
National Research University Higher School of Economics

 Eduard Babkin – Russia
National Research University Higher School of Economics

 Svetlana Chuprina – Russia
Perm State National Research University

 Pavel Drobintsev – Russia
Saint-Petersburg State Polytechnic University

 Liliya Emaletdinova – Russia
Institute for Technical Cybernetics and Informatics, KNRTU

 Victor Gergel – Russia
Lobachevsky State University of Nizhny Novgorod

 Efim Grinkrug – Russia
National Research University Higher School of Economics

 Maxim Gromov – Russia
Tomsk State University

 Vladimir Hahanov – Ukraine
Kharkov National University of Radioelectronics

 Shihong Huang – USA
Florida Atlantic University

 Iosif Itkin – Russia
Exactpro Systems

 Alexander Kamkin – Russia
Institute for System Programming of RAS

 Vsevolod Kotlyarov – Russia
Saint-Petersburg State Polytechnic University

 Oleg Kozyrev – Russia
National Research University Higher School of Economics

 Vladimir Kozyrev – Russia
National Research Nuclear University "MEPhI"

 Daniel Kurushin – Russia
State National Research Polytechnic University of Perm

 Peter Gorm Larsen – Denmark
Aarhus University

 Rustam Latypov – Russia
Institute of Computer Science and Information Technologies, KFU

 Alexander Letichevsky – Ukraine
Glushkov Institute of Cybernetics, NAS

 Alexander Lipanov – Ukraine
Kharkov National University of Radioelectronics

 Irina Lomazova – Russia
National Research University Higher School of Economics

 Ludmila Lyadova – Russia
National Research University Higher School of Economics

 Victor Malyshko – Russia
Moscow State University

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

 Tiziana Margaria – Germany
University of Potsdam

 Marek Miłosz – Poland
Institute of Computer Science, Lublin University of Technology

 Alexey Namestnikov – Russia
Ulyanovsk State Technical University

 Valery Nepomniaschy – Russia
Ershov Institute of Informatics Systems

 Mykola Nikitchenko – Ukraine
Kyiv National Taras Shevchenko University

 Yuri Okulovsky – Russia
Ural Federal University

 Elena Pavlova – Russia
Microsoft Research

 Ivan Piletski – Belorussia
Belarusian State University of Informatics and Radioelectronics

 Vladimir Popov – Russia
Ural Federal University

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

 Rustam Sabitov – Russia
Kazan National Research Technical University

 Nikolay Shilov – Russia
Ershov Institute of Informatics Systems

 Ruslan Smelyansky – Russia
Moscow State University

 Valeriy Sokolov – Russia
Yaroslavl Demidov State University

 Petr Sosnin – Russia
Ulyanovsk State Technical University

 Veniamin Tarasov – Russia
Povolzhskiy State University of Telecommunications and Informatics

 Sergey Ustinov – Russia
Saint-Petersburg State Polytechnic University

 Vladimir Voevodin – Russia
Research Computing Center of Moscow State University

 Dmitry Volkanov – Russia
Moscow State University

 Mikhail Volkov – Russia
Ural Federal University

 Nadezhda Yarushkina – Russia
Ulyanovsk State Technical University

 Rostislav Yavorsky – Russia
National Research University Higher School of Economics

 Nina Yevtushenko – Russia
Tomsk State University

 Vladimir Zakharov – Russia
Moscow State University

 Sergey Zaydullin – Russia
Kazan National Research Technical University

Organizing Committee Chairs and Secretaries

 Igor Chernorutskiy – Russia
Saint-Petersburg State Polytechnic University

 Tatyana Elamic – Russia
Saint-Petersburg State Polytechnic University

 Alexander Kamkin – Russia
Institute for System Programming of RAS

 Alexander Petrenko – Russia
Institute for System Programming of RAS

 Vsevolod Kotlyarov – Russia
Saint-Petersburg State Polytechnic University

Referees

Jean-Michel Adam

Sergey Avdoshin

Eduard Babkin

Mikhail Chupilko

Svetlana Chuprina

Pavel Drobintsev

Denis Efremov

Victor Gergel

Efim Grinkrug

Maxim Gromov

Shihong Huang

Iosif Itkin

Dmitry Ivankov

Anna Kalenkova

Alexander Kamkin

Dmitry Kosolobov

Vsevolod Kotlyarov

Artem Kotsynyak

Vladimir Kozyrev

Daniel Kurushin

Peter Gorm Larsen

Alexander Lipanov

Irina Lomazova

Lyudmila Lyadova

Victor Malyshko

Tiziana Margaria

Ivan Mikhailov

Alexey Namestnikov

Mykola Nikitchenko

Yuri Okulovsky

Nikolay Pakulin

Elena Pavlova

Alexander Petrenko

Ivan Piletski

Vladimir Popov

Yury Rogozov

Nikolay Shilov

Sergey Smolov

Valeriy Sokolov

Petr Sosnin

Veniamin Tarasov

Andrei Tatarnikov

Andrey Terekhov

Dmitry Volkanov

Mikhail Volkov

Nadezhda Yarushkina

Rostislav Yavorskiy

Nina Yevtushenko

Vladimir Zakharov

Modular construction of Time Petri net reachability graph

Ilona Knizhnikova

National Research University Higher School of Economics
iknizhnikova@gmail.com

Leonid Dworzanski

National Research University Higher School of Economics
leo@mathtech.ru

Abstract—Time Petri nets are an extension of Petri nets formalism with time specifications on transitions. The formalism is convenient to model distributed systems and enables capturing the time characteristics of distributed system activities. The primary tool for models behaviour understanding is reachability graph. In [10] the algorithm for constructing Time Petri net reachability graph was suggested. It is based on *essential* states, but the number of states in the resultant net reachability graph increases when time specification are scaled up, while the behaviour of the net is invariant under time specification scaling. We study the modification of this algorithm that allows to build Time Petri nets reachability graphs more efficiently using common divisors of the time specification in the components of a Time Petri net.

Keywords—time petri nets, reachability graph, essential states.

I. INTRODUCTION

Petri nets are a popular formalism for modelling concurrent systems. Different extensions of Petri nets and their applications are extensively studied in the literature [12], [3], [2], [11], [4], [6]. Obviously, time is the very important aspect of systems behaviour. Time restrictions like “this action can take from N to M seconds” are crucial for real-time system, net protocols, control systems et cetera. Time can be introduced into the Petri nets formalism in many different ways [1]. Moreover, even timeless distributed systems are hard to understand [8], [5]. There are two popular types of such nets: time Petri nets and timed Petri nets. Both of these modifications can simulate counter machines, i.e. are Turing-complete. Both can be used to model systems with time specifications, but in this article only time Petri nets are considered.

Time restrictions make behaviour of Time Petri nets models extremely hard to understand. It means that we have no options but to use computer aided means to check the correctness of a developing system or to analyze already constructed one. The crucial tool to understand the behaviour of a model is reachability graph. The problem of time Petri nets model checking can be solved via essential-states-based algorithm for constructing reduced reachability graphs [9]. But when all time specifications of a Time Petri net are multiplied by a constant, the size of reachability graph is increased or decreased while the behaviour of the net has not changed. In this work we study how to use this property of the algorithm to reduce the space requirements for analysis of a Time Petri net.

The paper is organized as follows. To start with, we provide basic notations of Petri nets. Then we define the Time Petri nets formalism. Then we provide a Time Petri net example which captures the process of a university course from the student

viewpoint. After that we apply our modification to build the reachability graphs of the net components of the example and provide the obtained results. The paper ends with conclusion.

II. PRELIMINARIES

Petri nets are a well known formalism widely used to model concurrent systems. Petri nets offer graphical notation and rigorous formal semantics. A Petri net is a marked directed bipartite graph, where the structure of the graph defines the behaviour of the model while the marking of the graph defines the current state.

Definition 2.1: A Petri (P/T-net) net is a 4-tuple (P, T, F, W) where

- P and T are disjoint finite sets of places and transitions, respectively;
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs;
- $W : F \rightarrow \mathbb{N}$ — an arc multiplicity function, that is a function which assigns every arc a positive integer number (arc multiplicity).

Following extension of W we denote as \widetilde{W} :

$$\widetilde{W}(x, y) = \begin{cases} n, & xFy \wedge W(x, y) = n \\ 0, & \neg xFy \end{cases}$$

A marking of a Petri net (P, T, F, W) is a multiset over P , i.e. a mapping $M : P \rightarrow \mathbb{N}$. By $\mathcal{M}(N)$ we denote a set of all markings of a P/T-net N . We say that the transition t in a P/T-net $N = (P, T, F, W)$ is active in the marking M iff for every $p \in \{p | (p, t) \in F\} : M(p) \geq \widetilde{W}(p, t)$. An active transition may fire, resulting in a marking M' such that $\forall p \in P : M'(p) = M(p) - \widetilde{W}(p, t) + \widetilde{W}(t, p)$.

Another notion we will use is hammock. While it is well known notion we will recall it informally. Hammock is a part of the graph such, that only two vertexes of hammock are linked with the rest of the graph. These vertexes are called starting and ending vertexes of hammock. There can be the subgraph of arbitrary complexity between these two vertexes, but this system should have no link to the graph except starting and/or ending vertex of hammock. [7]

III. MOTIVATING EXAMPLE

In this section, we provide the example of a time Petri net $Tnet$ that models the flow of some course.

The first transition of the *Tnet* denotes the beginning of a semester and the beginning of a course. The process consists of two almost independent scenarios, each of whose is represented by a separate net hammock. First of them (starting with the place *start1* and ending with *end1*) can be interpreted as the process of preparing for exams and working with a teacher. Second one models an examination process. The course can be completed iff these parts are performed successfully, i.e. final mark depends on both of them. But they do not block each other — a student has to pass the exam without regard to his work during the semester.

The process starts with a transition *course started* firing, which adds tokens to the places *start1* and *start2*. Then two hammocks are performing independently.

We start with considering the upper hammock. Firstly, transition *start solve problems* fires. It represents the beginning of student's work and adds tokens to the places *student* and *no more problems*. When both of these places contain tokens — student isn't solving a problem and is ready for actions. The course can be ended through firing the transition *all problems solved*, or the student can get some new problems to solve (transition *hand-in problem*). In the second case, the student needs to solve the problem. He or she starts with meeting with a tutor (going to place *meeting with tutor*), discusses the problem (transition *discussion with tutor* moves token to the place *student* and this enables transition *start work*) and then student starts working on it, and comes to some decision. (Transition *start work* fires and moves token to the place *solving*). After that, the problem is technically solved and transition *problem solved* fires adding token to the place *no more problems*, but the student still needs to meet his tutor again to discuss the result. (Chain *meeting with tutor–discussion with tutor*–adding token to place *student* fires again). Only after that student understands the nuances of the problem and the solution well enough.

Then the cycle may repeat again — student ends the process or gets a new problem. If the first case has place, the hammock finishes its execution.

Now we consider the second hammock. This hammock starts with transition *send assignment* firing adding tokens to places *solving assignment* and *submission opened*. Then the hypothetical student has two options: complete his work before deadline (transition *submission*), let the system register his work (transition *deadline*), and then just get his mark by getting through transition *evaluate work*.

If student had not managed to pass his work in time, the system registers this (transition *deadline*) and the submission is closed (disabling transition *submission*). Then he or she has no choice, but to pass the work behind time (transition *commission arranged*), wait for a re-examination (place *preparing for commission*) and go through it (transition *commission*). Independently of the success or the failure of his examination, the student gets his mark — would it be A or F at the transition *evaluate work* firing.

We will not provide the formal definition of a workflow net here, but this model is a sound workflow net, i.e. initial marking has one token in the *start* place. When a token reaches the *end* place there are no other tokens in the net. And the marking with the *end* place marked can always be reached.

IV. TIME PETRI NETS

In this section we define Time Petri nets (TPN).

We will use the definition of Time Petri net as given in [10]. Time Petri net (TPN) is an extended classic Petri net where each transition t is associated with a time interval $[a_t, b_t]$. If t is enabled, it still cannot fire before a_t time units have elapsed, and it has to fire no later than b_t time units after being enabled, unless it has meanwhile become disabled by the firing of another transition.

Time counting is started from the moment of enabling t and is reset if t has been disabled. a_t is called "earliest firing time" of t (short $eft(t)$) and b_t is "latest firing time" of t (short $lft(t)$). The firing of a transition does not take any time.

The interval bounds are non-negative rational numbers, and b_t can also be ∞ .

Definition 4.1: A Time Petri net (TPN) is a 6-tuple $Z = (P, T, F, V, m_0, I)$ such that

- 1) the 5-tuple $S(Z) = (P, T, F, V, m_0)$ is a Petri net,
- 2) $I : T \rightarrow Q_0^+ \times (Q_0^+ \cup \infty)$ and for each $t \in T$, with $I(t) = (I1(t), I2(t))$ it holds that $I1(t) \leq I2(t)$

Here $I(t)$ is the time interval of the transition t , during this interval t is ready to fire, $I1(t)$ is $eft(t)$ and $I2(t)$ is $lft(t)$.

Definition 4.2: (p -marking) Let P be the set of all places in a Time Petri net Z . A p -marking in Z is a (total) function $m : P \rightarrow \mathbb{N}$.

Definition 4.3: (t -marking) Let T be the set of all transitions in a time Petri net Z . Any (total) function $h : T \rightarrow \mathbb{R}_0^+ \cup \#$ is a t -marking in Z .

Definition 4.4: Let $Z = (P, T, F, V, m_0, I)$ be a Time Petri net, m a p -marking and h a t -marking in Z . A state in Z is a pair $z := (m, h)$ such that

- 1) $\forall t((t \in T \wedge t^- > m) \rightarrow h(t) = \#)$.
- 2) $\forall t((t \in T \wedge t^- \leq m) \rightarrow h(t) \in \mathbb{R}_0^+ \wedge h(t) \leq lft(t))$.

Claim 1. Time specification of transitions of an arbitrary time Petri net can be required to be non-negative integers without loss of generality [10]

V. REACHABILITY GRAPH

Firstly we consider original algorithm proposed in [10]. It is based on the conception of essential states.

Definition 5.1: (modified state change) Let τ be a non-negative real number and $z = (m, h)$ be a state in the Time Petri net Z . It is possible for time τ to elapse in the state z in Z if

$$\forall(t \in T \wedge h(t) \neq \# \rightarrow h(t) + \tau \leq lft(t))$$

The elapsing of time τ will change z into the state $z' = (m', h')$ with

- 1) $m' := m$,

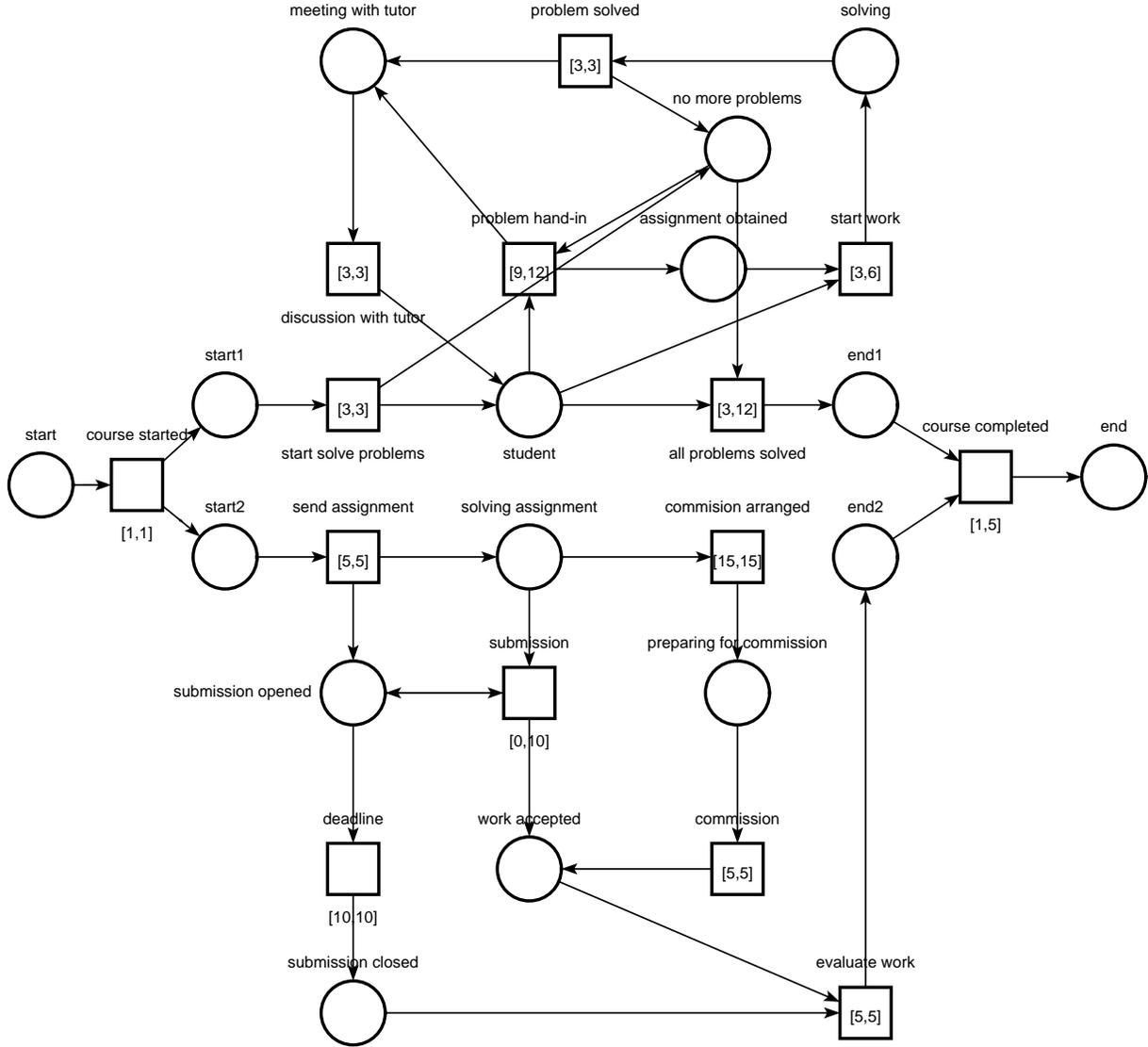


Fig. 1. The workflow net of a course process

2) $\forall t \in T \rightarrow$

$$h'(t) := \begin{cases} \# & \text{if } ift^- > m' \\ eft(t) & t^- \leq m' \wedge lft(t) = \infty \\ h(t) + \tau & \text{otherwise} \end{cases}$$

Definition 5.2: (essential-state) An integer-state $z = (m, h)$ in a Time Petri net Z is called essential-state when Z is defined with the modified firing rule.

Definition 5.3: (reachable essential-state) Let Z be an arbitrary Time Petri net. The set $REIS_Z$ of all reachable essential-states in Z is defined as follows:

$REIS_Z := \{z | z_0 \xrightarrow{\sigma(\tau)} z\}$, z is an essential-state and $\sigma(\tau)$ is a run feasible in Z

Set of all reachable essential states of the net carries complete information about this net's behavioural properties. This characteristic makes it possible to construct reachability graph, which includes only essential states (and is much smaller than original reachability graph) and still provides enough information to analyse behavioural properties of the net.

Definition 5.4: (reachability graph for arbitrary Time Petri net) Let $Z = (P, T, F, V, m_0, I)$ be an arbitrary (finite or infinite) Time Petri net with $T = t_1, \dots, t_n$. The (reduced) reachability graph $RG_Z^{redu} := (W, E, L)$ of Z is the directed graph with edge labels whose set of vertices W , set of edges E and edge labels from $L \subseteq N \times T$ are defined by Algorithm 1.

Algorithm 1: Time Petri net reachability graph

```
begin
  R := Z0; W := ∅; E := ∅;
  while R ≠ ∅ do
    select z = (m, h) from R; R := R - {z};
    W := W ∪ {z};
    if {t ∈ T | t- ≤ m} ≠ ∅ then
      if {t ∈ T | t- ≤ m ∧ lft(t) ≠ ∞} ≠ ∅ then
        Let k := min{lft(t) - h(t) | t- ≤ m}
        else
          Let
            k := max{eft(t) - h(t) | t- ≤ m}
      for time = 0 to k do
        for i = 1 to n do
          if ti ready to fire in (m, h) + time then
            Let z' be such that z  $\xrightarrow{time\ t_i}$  z';
            E := E ∪ {(z, [time, ti], z')};
            if z' ∉ W then
              R := R ∪ {z'}
```

VI. MODULAR CONSTRUCTION OF TIME PETRI NET REACHABILITY GRAPH

Algorithm constructing reachability graph through essential states demonstrates high performance, when it works with nets, whose time intervals are not very big. But if we multiply all time specifications in a net on the same number, number of essential states increases dramatically. But such scaling does not affect behavioural properties of the net and the set of reachable p-markings of the scaled and the original nets are the same.

There is an evident way to fix the problem — just check if the net can be down-scaled (divided by greatest common divisor) and than analyse this downscaled net. But what if we have the net consisting of two independent components. Both of these components can be down-scaled, if considered independently, but the time specification of these components contains coprime numbers.

Such net can have huge reachability graph, which can not be constructed because of the time or memory restrictions. But the graphs for each of these parts can easily be built via the downscale procedure. And these graphs carry enough information to analyse behavioural properties of the original net, such as boundedness, for example. We propose the way to solve the problem through searching for the detached hammocks, finding greatest common divisor (GCD) for all natural time specification inside them, dividing these specifications on the GCD and than applying reachability graph construction algorithm to each of the found hammocks.

Obtained reachability graphs can be used for performing model checking upon each of them and deduce identified properties for the whole net, but for the moment we consider only receiving reachability graphs.

We applied this technique to our example. The example contains two hammocks. First includes all vertexes and tran-

Algorithm 2: Modular construction of Time Petri net reachability graph

```
begin
  Let H be the set of hammocks in the net;
  X := ∅;
  foreach h ∈ H do
    foreach Transition t ∈ h do
      Insert eft(t) into X;
      Insert lft(t) into X;
    GCDh = Find GCD(X);
    foreach Transition t ∈ h do
      T'h = {t' | t ∈ Th ∧ eft(t') =
        eft(t)/GCDh ∧ lft(t') = lft(t)/GCDh};
      h' = (Ph, T'h, F, V)
    Apply Algorithm 1 to h
```

sitions between places *start1* and *end1* inclusively. Second starts at place *start2* and ends with *end2*. We can predict the successful application of the modular approach as the GCD of the first hammock time specifications is 3 and the GCD of the second hammock is 5, which are coprime numbers. Then we applied 2 and received the following results Table I.

The reachability graph of the whole net consists of 208 vertexes and 907 edges. But if we split up the net into 2 hammocks and then construct reachability graph for each of them, then the sum of them will be much lesser than the reachability graph of the whole net. In our case, the original net's reachability graph is not very big, because it is just an example, but even here reduction is significant.

TABLE I. ORIGINAL AND MODIFIED ALGORITHMS COMPARISON

	Vertexes	Edges
Original (full net)	208	907
Modified (hammock 1)	8	14
Modified (hammock 2)	7	9

VII. CONCLUSION

In this paper we provided an example Time Petri net that models the process of an academic course. The numeric properties of the transitions time specifications and our modification of the algorithm enabled us decrease time and space requirements for the analysis of the net. We constructed a reachability graph that is much smaller than the original one. The further research directions are to characterize the class of time Petri nets and the behavioural properties for which such technique is applicable.

ACKNOWLEDGMENT

This study was carried out within the National Research University Higher School of Economics' Academic Fund.

REFERENCES

- [1] B. Bérard, F. Cassez, S. Haddad, D. Lime, and O. H. Roux. Comparison of different semantics for time petri nets. In *ATVA*, pages 293–307, 2005.

- [2] B. Berthomieu, M. Boyer, and M. Diaz. Time petri nets. In *Petri Nets*, pages 123–161.
- [3] H. Genrich and K. Lautenbach. The Analysis of Distributed Systems by means of Predicate/Transition-Nets. In G. Kahn, editor, *Semantics of Concurrent Compilation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146. Springer-Verlag, Berlin, 1979.
- [4] K. Jensen and L. M. Kristensen. *Coloured Petri nets: modelling and validation of concurrent systems*. Springer, 2009.
- [5] L. Lamport. What good is temporal logic? In *IFIP Congress*, pages 657–668, 1983.
- [6] I. A. Lomazova. Nested petri nets for adaptive process modeling. In *Pillars of computer science*, pages 460–474. Springer, 2008.
- [7] I. A. Lomazova. Interacting workflow nets for workflow process re-engineering. *Fundamenta Informaticae*, 101(1):59–70, 2010.
- [8] S. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
- [9] L. Popova-Zeugmann. *Essential states in time Petri nets*. Citeseer, 1998.
- [10] L. Popova-Zeugmann. *Time Petri Nets*. Springer, 2013.
- [11] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. 1974.
- [12] M. Schiffers and H. Wedde. Analyzing program solutions of coordination problems by cp-nets. In J. Winkowski, editor, *Mathematical Foundations of Computer Science 1978*, volume 64 of *Lecture Notes in Computer Science*, pages 462–473. Springer Berlin Heidelberg, 1978.

On the deadlock control in parallel resource-constrained workflows

Vladimir A. Bashkin
and Nadezhda Yu. Panfilova
Yaroslavl State University
Yaroslavl, Russia 150000

Email: v_bashkin@mail.ru lillian007@mail.ru

Abstract—We study the verification of the soundness property for workflow nets extended with resources. A workflow is sound if it terminates properly (no deadlocks and livelocks are possible). A class of resource-constrained workflow nets (RCWF-nets) is considered, where resources can be used by a process instance, but cannot be created or spent.

Two sound RCWF-net, using the same set of resources, can be put in parallel. This parallel composition in some cases may produce additional deadlocks. A problem of deadlock avoidance in parallel workflows is studied, some methods of deadlock search and control are presented.

Keywords—Petri net, workflow, soundness, deadlock, RCWF-net, parallel composition

I. INTRODUCTION

Workflow management systems provide the automated support and coordination of business and technological processes to reduce costs and flow times and to increase quality of service and productivity. Workflows orchestrate people, resources, technology and information flow. Workflow nets [1], [2], a particular class of Petri nets, have become one of the standard ways to model and analyze workflow processes.

Workflow net is an abstraction of the workflow that can be used to check the so-called soundness property. This property guarantees the absence of livelocks, deadlocks, and other anomalies that can be detected without domain knowledge. Nowadays there exists a number of soundness notions (see [3] for a survey). Informally, the classical soundness ensures that from any reachable state the system may terminate properly.

A workflow consists of a set of coordinated tasks describing the flow of work within the organization. In real world the occurrence of those tasks may depend on resources, such as machines, manpower, and raw material. To take resources into account different extensions of a base formalism of WF-nets have been introduced, cursing different versions of soundness.

In [4], [5] a specific class of WFR-nets with decidable soundness was studied. In [10], [12] a more general class of Resource-Constrained Workflow Nets (RCWF-nets) was defined. Informally, the authors impose two constraints on resources. First, they require that all resources that are initially available are available again after terminating of all cases. Second, they also require that for any reachable marking, the number of available resources does not override the number of initially available resources.

In [6], [7] a more general case of arbitrary resource transformations was studied.

In [10] it was proven that for RCWF-nets with a single resource type generalized soundness can be effectively checked in polynomial time. In [12] it was proven that generalized soundness is decidable in RCWF-nets with an arbitrary number of resource places (by reducing to the home-space problem).

Although soundness is decidable, there is so far no efficient decision algorithm because the proposed algorithm decides a home-space property, which requires a finite but (in general) too high number of reachability checks [12]. In addition, the problem of the calculation of the smallest number of resources for which soundness can be proved, remains open.

In this paper we consider a compositional approach to this problem. We investigate possible ways of minimal resource partitioning in control-independent and resource-dependent parallel branches of a workflow. We define a natural notion of parallel composition of two RCWF-nets, sharing common resource places. Parallelism may introduce additional deadlocks here, but we prove that these deadlocks (and other soundness violations) are avoidable by an enlargement of the initial resource. We present an approach, that allows to compute a nontrivial subset of minimal sound resources of a decomposable RCWF-net.

The main result of the paper is a method of deadlock avoidance for parallel workflows. We show that under certain circumstances a composite workflow can be restructured in such a way that the resulting net would require not a sum but a union of minimal sound resources of its parallel subnets. This allows to save a significant part of resources without any violation of the soundness property.

The paper is organized as follows. In Section 2 basic definitions of multisets and Petri nets are given. In Section 3 resource-constrained workflow nets and their soundness properties are formally defined. In Section 4 we study reachability properties of sound RCWF-nets. In Section 5 a notion of parallel composition of RCWF-nets is introduced. Several result are formulated, describing how minimal resources of a composite workflow can be obtained from minimal resources of its parallel subnets. In Section 6 deadlock/livelock avoidance methods are presented. The first one can be applied for any pair of sound workflows, but requires specific run-time control, not incorporated into the net itself. The second one uses the

original Petri net structure, but is applicable to the safe nets only. Section 7 contains some conclusions.

II. PRELIMINARIES

Let S be a finite set. A *multiset* m over a set S is a mapping $m : S \rightarrow \text{Nat}$, where Nat is the set of natural numbers (including zero).

For two multisets m, m' we write $m \subseteq m'$ iff $\forall s \in S : m(s) \leq m'(s)$ (the inclusion relation). The sum and the union of two multisets m and m' are defined as usual: $\forall s \in S : m + m'(s) = m(s) + m'(s)$, $m \cup m'(s) = \max(m(s), m'(s))$. By $\mathcal{M}(S)$ we denote the set of all finite multisets over S .

Let P and T be disjoint sets of *places* and *transitions* and let $F : (P \times T) \cup (T \times P) \rightarrow \text{Nat}$. Then $N = (P, T, F)$ is a *Petri net*. A *marking* in a Petri net is a function $M : P \rightarrow \text{Nat}$, mapping each place to some natural number (possibly zero). Thus a marking may be considered as a multiset over the set of places. Pictorially, P -elements are represented by circles, T -elements by boxes, and the flow relation F by arcs. Places may carry tokens represented by filled circles.

For a transition $t \in T$ the *preset* $\bullet t$ and the *postset* $t \bullet$ are defined as the multisets over P such that $\bullet t(p) = F(p, t)$ and $t \bullet(p) = F(t, p)$ for each $p \in P$. Similarly, for a place $p \in P$ we define $\bullet p$ and $p \bullet$ as the multisets over T such that $\bullet p(t) = F(t, p)$ and $p \bullet(t) = F(p, t)$ for each $t \in T$.

A transition $t \in T$ is *enabled* in a marking M iff $\forall p \in P M(p) \geq F(p, t)$. An enabled transition t may *fire* yielding a new marking $M' =_{\text{def}} M - \bullet t + t \bullet$, i. e. $M'(p) = M(p) - F(p, t) + F(t, p)$ for each $p \in P$ (denoted $M \xrightarrow{t} M'$, or just $M \rightarrow M'$). We say that M' is *reachable* from M iff there is a sequence $M = M_1 \rightarrow M_2 \rightarrow \dots \rightarrow M_n = M'$. For a Petri net N by $\mathcal{R}(N, M_0)$ we denote the set of all markings reachable from its initial marking M_0 .

A net (N, M_0) is *bounded* iff $\mathcal{R}(N, M_0)$ is finite.

A net (N, M_0) is *safe* iff $\forall M \in \mathcal{R}(N, M_0), p \in P$ we have $M(p) \leq 1$. Places in safe nets can be considered as boolean variables (no tokens – false, 1 token – true).

III. RCWF-NETS

A resource-constrained workflow net (RCWF-net for short) is a tuple $N = (P_c, P_r, T, F_c, F_r, i, o)$ s.t.

- P_c is a finite set of control places;
- P_r is a finite set of resource places, $P_c \cap P_r = \emptyset$;
- T is a finite set of transitions, $P_c \cap T = P_r \cap T = \emptyset$;
- $F_c : (P_c \times T) \cup (T \times P_c) \rightarrow \text{Nat}$ is a multiset of control arcs;
- $F_r : (P_r \times T) \cup (T \times P_r) \rightarrow \text{Nat}$ is a multiset of resource arcs;
- $\forall t \in T \exists p \in P_c : F_c(p, t) + F_c(t, p) > 0$ (each transition is incident to some control place);
- $i \in P_c$ is a source place and $o \in P_c$ is a sink place (input and output), such that $\bullet i = o \bullet = \emptyset$;

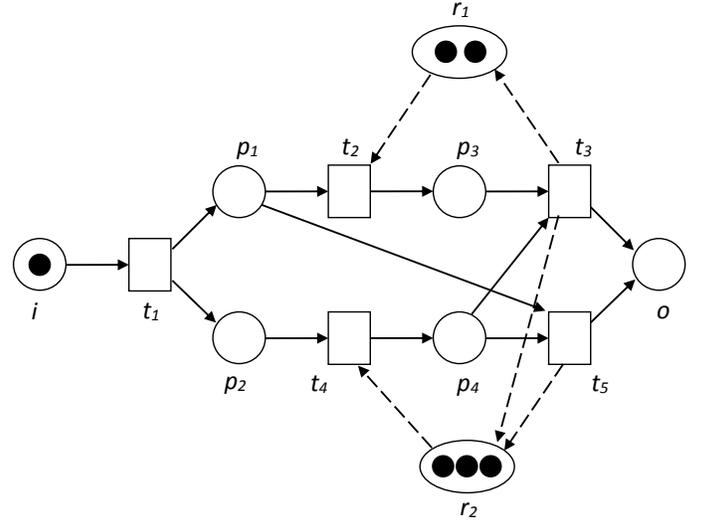


Fig. 1. RCWF-net

- every node from $P_c \cup T$ is on a path from i to o , and this path consists of nodes from $P_c \cup T$.

In RCWF-nets Petri net places are divided into control and resource ones. Note that all transitions are necessarily linked to control places — this guarantees the absence of “uncontrolled” resource modifications.

A marking is also divided into control and resource parts. For a multiset $c + r$, where $c \in \mathcal{M}(P_c)$ and $r \in \mathcal{M}(P_r)$, we write $c|r$.

For a net N a *resource* is a multiset over P_r . A *controlled resource* (a state) is a multiset over $P_c \cup P_r$.

Fig. 1 represents an example of a RCWF-net, where resource places r_1 and r_2 are depicted by ovals, resource arcs — by dotted arrows.

Every RCWF-net $N = (P_c, P_r, T, F_c, F_r, i, o)$ contains its *control subnet* $N_c = (P_c, T, F_c, i, o)$, which forms a RCWF-net with the empty set of resources.

A *marked net* $(N, i|r)$ is a net N together with some initial marking $i|r$ (here i denote a multiset, containing a single token in the input place i).

Let N be an RCWF-net. N is (r) -*sound* for some resource $r \in \mathcal{M}(P_r)$ iff $\forall c|r' \in \mathcal{R}(N, i|r)$ we have:

- 1) $r' \leq r$;
- 2) $o|r \in \mathcal{R}(N, c|r')$.

N is *sound* iff there exists some resource $r \in \mathcal{M}(P_r)$ such that N is (r') -sound for any $r' \geq r$.

For example, the net on Fig. 1 is sound, $(r_1 + r_2)$ -sound and not (r_1) -sound.

Thus soundness for an RCWF-net means, that, first, this workflow net can terminate properly from any reachable state, and, moreover, adding any extra resource does not violate the proper termination property.

In [12] it was proven that the soundness problem is decidable even in a more general case of multiple input tokens.

Definition 1: For a sound RCWF-net N by $\mathbf{res}(N)$ and $\mathbf{mres}(N)$ we denote the sets of sound and minimal sound resources:

- $\mathbf{res}(N) =_{\text{def}} \{r \in \mathcal{M}(P_r) \mid N \text{ is } (r + r') \text{ - sound for any } r' \in \mathcal{M}(P_r)\};$
- $\mathbf{mres}(N) =_{\text{def}} \{r \in \mathbf{res}(N) \mid \nexists r' \in \mathbf{res}(N) : r' < r\}.$

Obviously, $\mathbf{mres}(N)$ is finite.

For example, for the net on Fig. 1 we have $\mathbf{mres}(N) = \{r_1 + r_2\}$.

As it was stated in [12], the problem of finding $\mathbf{mres}(N)$ is still open. In this paper we introduce and evaluate a promising approach to this problem, based on the parallel composition/decomposition of RCWF-nets.

IV. PROPERTIES OF SOUND RESOURCES

The next statement formally defines a well-known “proper completion” property of sound workflows:

Fact 1: For any (r) -sound net N

$$c|r \in \mathcal{R}(N, i|r) \Rightarrow c = o \vee c \cap o = \emptyset.$$

Proof: Assume the converse: $o + m|r \in \mathcal{R}(N, i|r)$ for some non-empty m .

From the second requirement of soundness we have $o|r \in \mathcal{R}(N, o + m|r)$. However, since the place o doesn't have outgoing arcs, we have $\emptyset|r \in \mathcal{R}(N, m|r)$. But every transition in N has at least one *output* control place, thus $m = \emptyset$ — a contradiction. ■

Another established fact is the soundness and boundedness of the control subnet:

Fact 2: For any sound RCWF-net $N = (P_c, P_r, T, F_c, F_r, i, o)$ and its control subnet $N_c = (P_c, T, F_c, i, o)$ (that may be considered as an RCWF-net with an empty set of resources) we have:

- 1) N_c is (\emptyset) -sound;
- 2) $(N_c, i|\emptyset)$ is bounded;
- 3) if $c|\emptyset, c + c'|\emptyset \in \mathcal{R}(N, i|r)$ then $c' = \emptyset$.

Proof: (1) Assume the converse: N_c is not (\emptyset) -sound. Since N_c contains no resource places, only the second part of the soundness definition is violated:

$$\exists c|\emptyset \in \mathcal{R}(N_c, i|\emptyset) : o|\emptyset \notin \mathcal{R}(N_c, c|\emptyset).$$

Denote the corresponding transition sequence by σ (so we have $i|\emptyset \xrightarrow{\sigma} c|\emptyset$).

Now consider N . Obviously, there exists some large initial resource r s.t. $c|r' \in \mathcal{R}(N, i|c)$ for some r' — it is sufficient to sum all resources, required by transitions of σ .

On the other hand, no resource x would be enough to reach the final state $o|y$ from $c|x$ (for any y), since it is unreachable

even in the “resource-free” control subnet N_c . Hence N is not sound.

(2) Otherwise an infinite run is possible in $(N_c, i|\emptyset)$, containing an infinite number of different markings, and hence a pair of markings $c_1 < c_2$ with $i \rightarrow c_1 \rightarrow c_2 \rightarrow \dots$. From the soundness of N_c we have $c_1 \xrightarrow{\sigma} o$ for some sequence of transitions $\sigma \in T^*$. But from $c_1 < c_2$ the same sequence is possible in $c_2 : c_2 \xrightarrow{\sigma} o + (c_2 - c_1)$ — a contradiction to the proper completion property.

(3) Assume the converse. From the soundness property we have two transition sequences: $i|\emptyset \rightarrow c|\emptyset \rightarrow o|\emptyset$ and $i|\emptyset \rightarrow c + c'|\emptyset \rightarrow o|\emptyset$.

From the first sequence and the monotonicity of Petri nets we have $i + c'|\emptyset \rightarrow c + c'|\emptyset \rightarrow o + c'|\emptyset$. Combining with the second sequence, we obtain $i|\emptyset \rightarrow c + c'|\emptyset \rightarrow o + c'|\emptyset$ — a contradiction to the soundness property. ■

Every reachable $c \in \mathcal{M}(P_r)$ (a *control state* of a control subnet N_c) corresponds to a single reachable resource value:

Lemma 1: If N is sound, $r \in \mathbf{res}(N)$ and $c|r_1, c|r_2 \in \mathcal{R}(N, i|r)$, then $r_1 = r_2$.

Proof: Assume the converse: let $r_1 \neq r_2$.

Consider some $r' = r_1 + \delta_1 = r_2 + \delta_2$. From $r_1 \neq r_2$ we have $\delta_1 \neq \emptyset$ or $\delta_2 \neq \emptyset$ or both. Additionally, $\delta_1 \neq \delta_2$.

We have $i|r \rightarrow c|r_1 \rightarrow o|r$ and hence (from the monotonicity of Petri nets) $i|r + \delta_1 \rightarrow c|r_1 + \delta_1 \rightarrow o|r + \delta_1$. Similarly, $i|r + \delta_2 \rightarrow c|r_2 + \delta_2 \rightarrow o|r + \delta_2$. But $r_1 + \delta_1 = r' = r_2 + \delta_2$ and hence we have $i|r + \delta_1 \rightarrow c|r' \rightarrow o|r + \delta_2$. From the $(r + \delta_1)$ -soundness property it should be $\delta_1 = \delta_2$ — a contradiction. ■

Note that we cannot replace in the statement of Lemma 1 “ $r \in \mathbf{res}(N)$ ” by “ N is (r) -sound”, because an (r) -sound net is not necessarily $(r + \delta)$ -sound.

For (r) -soundness we have a weaker property:

Lemma 2: If N is (r) -sound and $c|r_1, c|r_2 \in \mathcal{R}(N, i|r)$, then

$$r_1 \not\prec r_2 \text{ and } r_1 \not\succeq r_2.$$

Proof: Similar to the previous Lemma. Assume the converse: $r_1 < r_2$ and hence $r_2 = r_1 + \delta_1$ with $\delta_1 \neq \emptyset$.

We have $i|r \rightarrow c|r_1 \rightarrow o|r$ and $i|r \rightarrow c|r_2 = c|r_1 + \delta_1 \rightarrow o|r$, and so $c|r_1 + \delta_1 \rightarrow o|r + \delta_1$ — a contradiction to the (r) -soundness. ■

Since any set of incomparable vectors over $\text{Nat}^{|P_r|}$ is finite, we have an obvious

Corollary 1: If N is (r) -sound then $\mathcal{R}(N, i|r)$ is finite.

A particular consequence of Lemma 1 is an inability of a cycle to modify a resource:

Proposition 1: If N is sound, $r \in \mathbf{res}(N)$, $c|r_1 \in \mathcal{R}(N, i|r)$ and $c|r_2 \in \mathcal{R}(N, c|r_1)$, then $r_1 = r_2$.

Proof: Immediately from Lemma 1. ■

Moreover, a sound net can perform only fixed resource transformations:

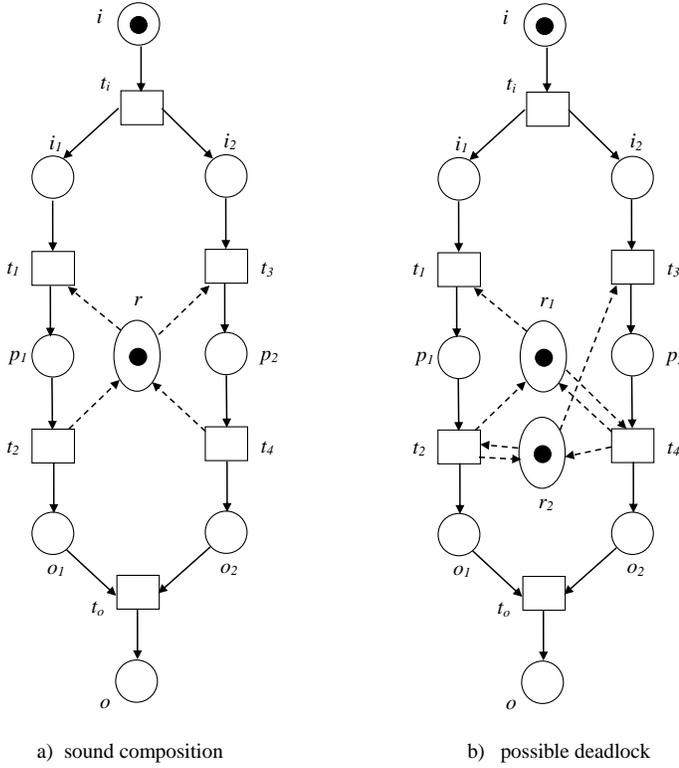


Fig. 2. Two examples of RCWF-nets compositions

Proposition 2: If N is sound, $r \in \mathbf{res}(N)$, $c|r' \in \mathcal{R}(N, i|r)$ and $u \in \mathcal{M}(P_r)$, then for any $c|v \in \mathcal{R}(N, i|r+u)$ we have $v = r' + u$.

Proof: Assume the converse. Hence $c|v, c|r' + u \in \mathcal{R}(N, i|r+u)$ with $v \neq r' + u$ — a contradiction to Lemma 1. ■

V. COMPOSITIONS OF RCWF-NETS

Nets with the same sets of resource places can be composed in parallel:

Definition 2: Let N_1 and N_2 be RCWF-nets with

- $N_1 = ((P_c)_1, P_r, T_1, (F_c)_1, (F_r)_1, i_1, o_1)$ and
- $N_2 = ((P_c)_2, P_r, T_2, (F_c)_2, (F_r)_2, i_2, o_2)$.

A *parallel composition* of N_1 and N_2 (denoted by $N = N_1 \parallel N_2$) is an RCWF-net $N = (P_c, P_r, T, F_c, F_r, i, o)$ with

- $P_c =_{\text{def}} (P_c)_1 \cup (P_c)_2 \cup \{i, o\}$,
- $T =_{\text{def}} T_1 \cup T_2 \cup \{t_i, t_o\}$,
- $F_c =_{\text{def}} (F_c)_1 \cup (F_c)_2 \cup \{(i, t_i), (t_i, i_1), (t_i, i_2), (t_o, o), (o_1, t_o), (o_2, t_o)\}$,
- $F_r =_{\text{def}} (F_r)_1 \cup (F_r)_2$.

We put two workflows in parallel, adding common source and sink places.

Examples of simple RCWF-nets compositions are given on Fig. 2. In the case Fig. 2.(a) both subnets has the same

minimal sound resource r , and the composition is also sound with this resource. The case Fig. 2(b) is quite different. Note that $r_1 + r_2$ is a minimal sound resource for both subnets, but the composition is not $(r_1 + r_2)$ -sound because of a deadlock $p_1 + p_2 | \emptyset$, reachable from $i|r_1 + r_2$. Any larger resource is sound.

Soundness of a resource for a subnet does not necessarily imply it's soundness for a composition (as one would expect, taking into account the conservativeness of resource transformations in an RCWF-net). A parallelism may introduce additional deadlocks. However, a simple kind of additive closure exists:

Theorem 1: If N_1 and N_2 are sound then $N_1 \parallel N_2$ is sound and, moreover:

- 1) $r_1 \in \mathbf{res}(N_1), r_2 \in \mathbf{res}(N_2) \Rightarrow r_1 + r_2 \in \mathbf{res}(N_1 \parallel N_2)$;
- 2) $r \in \mathbf{res}(N_1 \parallel N_2) \Rightarrow \exists r_1 \in \mathbf{res}(N_1) : r \leq r_1$;
- 3) $r \in \mathbf{mres}(N_1 \parallel N_2) \Rightarrow \exists r_1 \in \mathbf{res}(N_1) : r \leq r_1$.

Proof: The soundness itself and the first statement follows from Proposition 2. Note that subnets N_1 and N_2 here work independently, without interfering into each other's "part" of the common resource.

To prove the second statement we can take $r_1 = r$: since a resource is sound for a parallel composition, it properly supports system runs of the form $i|r \rightarrow i_1 + i_2|r \rightarrow o_1 + i_2|r \rightarrow o_1 + o_2|r$.

The third statement is a trivial consequence of the second one. ■

The first statement of Theorem 1 implies that

Corollary 2: If N_1 and N_2 are sound and $r_1 \in \mathbf{mres}(N_1), r_2 \in \mathbf{mres}(N_2)$, then there exists $r \in \mathbf{mres}(N_1 \parallel N_2)$ such that $r \leq r_1 + r_2$.

So, to find some minimal resource r one may search through a finite number of resources, less then or equal to $r_1 + r_2$. For every candidate $r' \leq r_1 + r_2$ the set $\mathcal{R}(N, i|r')$ is finite (Corollary 1) and can be constructed by a finite number of steps.

Note that we have not proven that this method of minimal sound resources computation allows to compute ALL elements of $\mathbf{mres}(N_1 \parallel N_2)$ (however, we believe it does). Nevertheless, the computed subset is always nonempty and nontrivial.

So, a problem of $\mathbf{mres}(N)$ calculation can be partially reduced to the same problem for subnets, composed in parallel. In most cases the process of decompositions ends with a purely sequential workflows, which may have a very simple set of sound (and minimal sound) resources.

VI. SOUNDNESS ENSURING

In this section we consider a resource r , sound for both subnets but not sound for a parallel composition (like $r_1 + r_2$ in Fig. 2(b)). Note that such a resource always enables a non-empty set of "good" runs (at least two: $i|r \rightarrow i_1 + i_2|r \rightarrow o_1 + i_2|r \rightarrow o_1 + o_2|r$ and $i|r \rightarrow i_1 + i_2|r \rightarrow i_1 + o_2|r \rightarrow o_1 + o_2|r$). Hence a resource is not worthless and it would be interesting to develop some control policies or system

transformations, preserving all “good” runs and disabling all “bad” ones (without increasing the initial resource).

So we consider both kinds of possible undesirable (not properly terminating) behaviors of a Petri net, namely, deadlocks and livelocks.

A reachable marking $c|r$ is a *deadlock state* iff $c \neq o$ and there is no transition $t \in T$ s.t. $c|r \xrightarrow{t} c'|r'$ for some c', r' .

A finite set L of reachable markings is a *livelock* iff

- 1) $|L| > 1$;
- 2) for any $c|r, c'|r' \in L$ there is a finite transition sequence $\sigma \in T^*$ s.t. $c|r \xrightarrow{\sigma} c'|r'$;
- 3) for any $c|r \in L$ and $t \in T$ s.t. $c|r \xrightarrow{t} c''|r''$ we have $c''|r'' \in L$.

A *livelock state* is a state that belongs to some livelock.

Note that by definition $o|r \notin L$ for any r .

By $D(N, i|r)$ we denote a set of all deadlock and livelock states of a marked RCWF-net $(N, i|r)$.

Theorem 2: If $N = N_1 || N_2$ and $r \in \text{res}(N_1) \cap \text{res}(N_2)$ then $(N, i|r)$ is bounded (i.e. $\mathcal{R}(N, i|r)$ is finite).

Proof: From the second statement of Fact 2 the sets of control markings are finite for both N_1 and N_2 . Obviously, the set of reachable control markings of N is a subset of a product of these two finite sets, hence it is also finite.

Now consider markings from $\mathcal{R}(N, i|r)$. Assume the converse — this set is infinite. Hence from the boundedness of the control subnet there exists some control cycle, strictly increasing the resource: $i|r \rightarrow c_1 + c_2|r' \xrightarrow{\sigma} c_1 + c_2|r' + r''$ with $c_1 \in \mathcal{M}((P_c)_1), c_2 \in \mathcal{M}((P_c)_2), \sigma \in T^*$ and $r'' \neq \emptyset$.

Recall that $T = T_1 \cup T_2$ and denote by σ_1 and σ_2 the largest subsequences of σ s.t. $\sigma_1 \in (T_1)^*$ and $\sigma_2 \in (T_2)^*$. Obviously, σ_1 and σ_2 are control cycles in N_1 and N_2 respectively.

From Proposition 1 neither σ_1 nor σ_2 can change the resource, hence their composition also cannot do this — a contradiction. ■

Since $D(N, i|r) \subseteq \mathcal{R}(N, i|r)$ we have:

Corollary 3: If $N = N_1 || N_2$ and $r \in \text{res}(N_1) \cap \text{res}(N_2)$ then $D(N, i|r)$ is finite.

So the set of deadlocks and livelocks is computable by a simple reachability set construction and search. A naive deadlock control policy would be to compute a set of all deadlocks/livelocks and all their predecesing states and to control them in run-time, not allowing a system to make the wrong “last step”.

A. Safe nets

A rather interesting case are safe workflows, i.e. RCWF-nets with safe control subnets (where none of the control places can accumulate more than one token). This is not a strong restriction, because every bounded net is weakly bisimilar to some safe net. Note that the net in Fig. 2(b) is safe and still has a deadlock.

A safe RCWF-net has only *ordinary* control arcs: $F_c(x, y) \leq 1$ for any x and y .

We can apply a transformation, eliminating all deadlocks/livelocks in a safe net:

Definition 3: Let N_1 and N_2 be sound safe RCWF-nets with the same set P_r of resource places, and let $r \in \mathcal{M}(P_r)$ be a resource s.t. $r \in \text{res}(N_1)$ and $r \in \text{res}(N_2)$.

Let $N = N_1 || N_2 = (P_c, P_r, T, F_c, F_r, i, o)$ be a not-(r)-sound parallel composition of N_1 and N_2 .

By $D_c(N, i|r)$ we denote the set of all different control parts of elements of $D(N, i|r)$, and let $Z = |D_c(N, i|r)|$ (obviously, we have $Z > 0$).

A net $(N_a, i|r+v)$, where $N_a = (P_c, P_r \cup V, T, F_c, F_r \cup F_{in} \cup F_{out}, i, o)$, is called a *controlled system* of $(N, i|r)$ iff

- $V = \{v_k | k \in \overline{1, Z}\}$ is a set of additional “holding” places, with their number equal to the number of possible control deadlocks/livelocks in the net N ;
- F_{in} are input holding arcs such that $F_{in} = \{(v_k, t) | F_c(t, p) = 1 \text{ for some } p \in d_k, \text{ where } d_k \text{ is a } k^{\text{th}} \text{ element of } D_c(N, i|r)\}$;
- F_{out} are output holding arcs such that $F_{out} = \{(t, v_k) | F_c(p, t) = 1 \text{ for some } p \in d_k, \text{ where } d_k \text{ is a } k^{\text{th}} \text{ element of } D_c(N, i|r)\}$;
- $v = (|d_1| - 1)v_1 + (|d_2| - 1)v_2 + \dots + (|d_Z| - 1)v_Z$, where d_k is a k^{th} element of $D_c(N, i|r)$.

Main idea to start with is that we have to avoid the consequent triggering of transitions leading to deadlock/livelock places. Thus for every element of $D(N, i|r)$ we include into the net a holding (restraining) place which resource will allow triggering of transitions leading to just one control place among them. Resource should be put back to holding place right after token leaves deadlock place. Implementation of this idea based on the net from Fig. 2 is shown on Fig. 3.

Note that the initial resource $(r+v)$ is actually not an increased original initial resource r , since v is built over the new (additional) set of resource places.

Theorem 3: Let N_1 and N_2 be sound safe RCWF-nets with the same set P_r of resource places, and let $r \in \mathcal{M}(P_r)$ be a resource s.t. $r \in \text{res}(N_1)$ and $r \in \text{res}(N_2)$.

Let $N = N_1 || N_2 = (P_c, P_r, T, F_c, F_r, i, o)$ be a not-(r)-sound parallel composition of N_1 and N_2 .

Let net $(N_a, i|r+v)$ be the controlled system of the marked net $(N, i|r)$. Then $(N_a, i|r+v)$ is $(r+v)$ -sound.

Proof: Obviously, none of the original deadlocks/livelocks is reachable (by construction).

Now we need to prove that no new deadlocks/livelocks are introduced.

Consider some deadlock/livelock $c_1 + c_2|r$ of the original net. From the third statement of Fact 2 and the safety property none of the control states of N_1 , except c_1 can have the same or larger total number of tokens in all places of c_1 (similarly for the net N_2 and places of c_2). Hence $|c_1 + c_2| - 1 = |c_1| + |c_2| - 1$

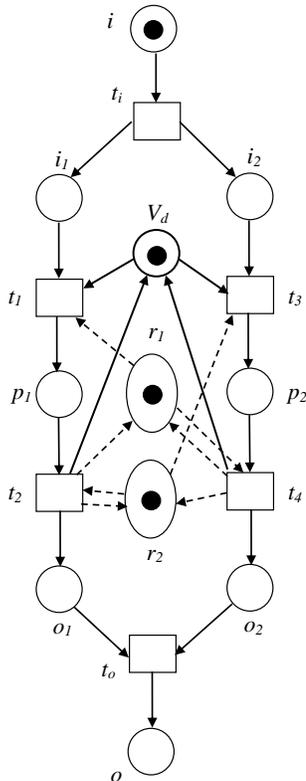


Fig. 3. Examples of RCWF-nets compositions with deadlock control

tokens is enough for all control states except this particular deadlock/livelock — hence the corresponding holding place would not introduce any undesirable restriction. ■

VII. CONCLUSION

We presented two methods of deadlock/livelock avoidance for a restricted resource. The first one can be applied for any pair of sound workflows, but requires specific run-time control, not incorporated into the net itself. The second one uses the original Petri net structure, but is applicable to the safe nets only. The proposed technique is similar to a technique, studied in the area of Flexible Manufacturing Systems (see [9] for a classical result). However, the key difference is the possibility of parallel behaviours in subnets (in FMS each subnet is a simple sequential automaton).

Further research will consider the application of holding places technique to the general case of RCWF-nets. As it can be seen from some preliminary counterexamples, the method would possibly require some additional modifications.

The problem of exact $mres(N)$ calculation is still open but it will be studied in the further research. We believe that our approach can be applied here, at least for large nontrivial subclasses of RCWF-nets. For example, we plan to study structured workflows [2], [8], that can be obtained from primitives by a set of algebraic operations, such as parallel and sequential compositions. Another interesting method of sound resource compositions was presented in [11] — based on algebraic expressions over multisets.

Our method is not implemented in applications. However, we believe it can be used as a verification and/or optimization tool in workflow management systems. Basically, it may enable the elimination of specific deadlocks and livelocks, induced by incorrect (unverified) parallel composition of submodules.

REFERENCES

- [1] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
- [2] W.M.P. van der Aalst, K.M. van Hee. *Workflow Management: Models, Methods and Systems*, MIT Press, 2002.
- [3] W.M.P. van der Aalst, K.M. van Hee, A.H.M. Hofstede, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, M.T. Wynn. Soundness of workflow nets: classification, decidability, and analysis, *Form. Asp. of Comp.*, 23(3):333–363, Springer, 2011.
- [4] K. Barkaoui, L. Petrucci. Structural Analysis of Workflow Nets with Shared Resources. In *Proc. Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM98)*, volume 98/7 of *Computing Science Reports*, pages 82–95, Eindhoven University of Technology, 1998.
- [5] K. Barkaoui, R. Ben Ayed, Z. Sbaï. Workflow Soundness Verification based on Structure Theory of Petri Nets. *International Journal of Computing and Information Sciences*, Vol. 5, No. 1, 2007. P.51–61.
- [6] V. A. Bashkin, I. A. Lomazova. Resource equivalence in workflow nets. In *Proc. Concurrency, Specification and Programming, 2006*, volume 1, pages 80–91. Berlin, Humboldt Universitat zu Berlin, 2006.
- [7] V.A. Bashkin, I.A. Lomazova. Soundness of Workflow Nets with an Unbounded Resource is Decidable *Joint Proc. of Petri Nets and Software Engineering (PNSE'13) and Modeling and Business Environments (ModBE'13). Milano, 2013*. Vol. 989 of CEUR. 2013. P. 61–75.
- [8] P. Chrzastowski-Wachtel. Sound Markings in Structured Nets. In *Proc. Concurrency, Specification and Programming, 2005*, pages 71–85. Warsaw, Warsaw University, 2005.
- [9] J. Ezpeleta, J.-M. Colom, J. Martinez. A Petri Net Based Deadlock Prevention Policy for Flexible Manufacturing Systems. *IEEE Transactions on Robotics and Automation*, 11(2), 1995. P.173–184.
- [10] K. van Hee, A. Serebrenik, N. Sidorova, M. Voorhoeve. Soundness of Resource-Constrained Workflow Nets. In *Proc. ICATPN 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2005.
- [11] I.A. Lomazova, I.V. Romanov. Analyzing Compatibility of Services via Resource Conformance. *Fundamenta Informaticae*, Vol. 128, No. 1–2, 2013. P.129–141.
- [12] N. Sidorova, C. Stahl. Soundness for resource-constrained workflow nets is decidable. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(3), 2013. P.724–729.

LTL-specification, verification and construction of PLC programs

Ryabukhin D. A.
Yaroslavl State University
Yaroslavl, Russia
Email: dmitriy_ryabukhin@mail.ru

Kuzmin E. V.
Yaroslavl State University
Yaroslavl, Russia
Email: kuzmin@uniyar.ac.ru

Abstract—An approach to specification, verification and construction of PLC programs for discrete problems is proposed. For the specification of the program behavior, we use the linear-time temporal logic LTL. Programming is carried out in ST, IL and LD languages according to an LTL-specification. The correctness analysis of an LTL-specification is carried out by the symbolic model checking tool Cadence SMV. A new approach to PLC-programming is shown by an example. For a discrete problem, we give an ST-program, its LTL-specification and an SMV-model.

I. INTRODUCTION

Application of programmable logic controllers (PLCs) for systems controlling complex industrial processes makes exacting correctness demands to PLC-programs. Any software error is considered to be inadmissible. However, the existing development tools for programming PLC, for example widely known CoDeSys (Controller Development System) [8], provide only usual debugging facilities through testing programs (not guaranteeing total absence of errors) by means of a visualization of PLC-control objects. At the same time certain theoretical knowledge and experience of applying the existing developments in the field of formal methods of modeling and analysis of software systems are accumulated. The programming of logical controllers is a practical area, in which existing developments could have successful application. Successful application is understood as implementation of formal methods in the process of program design at the level of a well-functioning technology which is clear to all specialists involved in this process — engineers, programmers and testers. Being as usual of a small size and having a finite state space, PLC-programs are exceptionally convenient objects for the formal (including automatic) analysis of correctness.

Programmable Logic Controllers (PLCs) are a special type of a computer widely used in automation systems [10], [5]. A PLC is a reprogrammable computer, based on sensors and actors, which is controlled by a user program. They are highly configurable and thus are applied to various industrial sectors. A PLC is a reactive system. A PLC repeats the execution of a user program periodically. There are three main phases for program execution (working cycle): 1) reading from inputs (sensors) and latching them in the memory, 2) program execution (with input variables remaining constant), 3) latching the values of the output variables to the environment.

Programming languages for logic controllers are defined by the IEC 61131-3 standard. This standard includes the description of five languages: SFC, IL, ST, LD and FBD.

IL (Instruction List) is an assembly language with an accumulator and jumps to labels. IL allows to work with any data types, to call functions and function blocks, written in IEC 61131-3 standard languages. IL is used to build small components, when critical control is required. Instructions are executed with the accumulator content. The IL accumulator is a universal container, which can keep values of any type.

ST (Structured Text) is high-level programming language. Its syntax is the adapted Pascal syntax.

LD (Ladder Diagram) represents a program by a graphical diagram based on circuit diagrams of relay logic hardware. The language itself can be seen as a set of connections between logical checkers (contacts) and actuators (coils). If a path can be traced between the left side of the connection and the output, through asserted contacts, the output coil storage bit is asserted true. If no path can be traced, the output is false.

This languages provides a possibility of application of all existing methods of program correctness analysis — testing, theorem proving [9] and model checking [7] — for verification of PLC-programs. Theorem proving is more applicable to “continuous” stability and regulation problems of the engineering control theory, since an implementation of these problems in PLC is associated with programming of an appropriate system of formulas. Model checking is most suitable for “discrete” problems of logical control, requiring PLCs with binary inputs and outputs. This provides a finite space of possible states of PLC-programs.

The most convenient for programming, specification and verification of PLC-programs are ST, LD and SFC languages, since they do not cause difficulties for neither developers nor engineers and can be easily translated into languages of software tools of automatic verification.

Earlier in the article [2], a review of methods and approaches to programming “discrete” PLC problems was carried out on languages LD, SFC and ST. For these approaches the usability of the model checking method for the analysis of program correctness with respect to the automatic verification tool Cadence SMV [13] was evaluated. Some possible PLC-program vulnerabilities arising at traditional approaches to programming of PLC was revealed. In particular, existing

articles relating to correctness analysis of PLC programs [3], [6], [11], [12] is mainly devoted to construction of translators from IEC 61131 standard languages to interface languages of verification software. Demonstration of results is carried out on trivial examples. However, our experience of working with the practical logic control problems showed that the direct translation does nothing for analysis of program properties, since it is often not possible to express desired properties in temporal logic languages.

In this article, an approach to construction and verification of PLC-programs for discrete problems is proposed. For the specification of the program behavior, we use the linear-time temporal logic LTL. Programming is carried out in ST, IL and LD languages according to an LTL-specification. The correctness analysis of an LTL-specification is carried out by the symbolic model checking tool Cadence SMV. A new approach to programming and verification of PLC-programs is shown by an example. For a discrete problem we give an ST-program, its LTL-specification and an SMV-model. The purpose of the article is to describe an approach to programming PLC, which would provide a possibility of PLC-program correctness analysis by the model checking method.

The further work is to build software tools for modeling, specification, construction and verification of PLC-programs.

II. MODEL CHECKING. A PLC PROGRAM MODEL

Model checking is the process of checking whether a given model (a Kripke structure) satisfies a given logical formula. A Kripke structure represents the behaviour of a program. A temporal logic formula encodes the property of the program. We use the linear-time temporal logic (LTL).

A *Kripke Structure* on a set of atomic propositions P is a state transition system $\mathcal{S} = (S, s_0, \rightarrow, L)$, with a non-empty set of states S , an initial state $s_0 \in S$, a transition relation $\rightarrow \subseteq S \times S$ which is defined for all $s \in S$, and a function $L : S \rightarrow 2^P$, labeling every state by a subset of atomic propositions.

A *Path* of the Kripke structure from the state s_0 — is an infinite consequence of states $\pi = s_0 s_1 s_2 \dots$ where $\forall i \geq 0$ $s_i \rightarrow s_{i+1}$.

The linear-time temporal logic language is considered as a specification language for behavioural properties of a programming model. PLC is a classic reactive control system, which once running must always have a correct infinite behavior. LTL formulas allow to represent this behavior.

The syntax of the LTL formula is given by the following grammar, $p_i \in P$:

$$\begin{aligned} \varphi, \psi ::= & true \mid p_0 \mid p_1 \mid \dots \mid p_n \mid \neg \varphi \mid \psi \wedge \varphi \mid \\ & X\varphi \mid \psi U \varphi \mid F\varphi \mid G\varphi. \end{aligned}$$

LTL formula describes a property of one path of the Kripke structure, descendant from an emphasized current state. The temporal operators X , F , G and U are interpreted as follows: $X\varphi$ — φ must hold at the next state, $F\varphi$ — φ must hold at some future state, $G\varphi$ — φ must hold at the current state and all future state, $\psi U \varphi$ — φ holds at the current or a future

state, and ψ must hold up until this point. In addition, classical logical operators \vee and \Rightarrow will be used further.

The Kripke structure satisfies an LTL formula (property) φ , if φ holds true for all paths, starting from the initial state s_0 .

The Kripke model for a PLC program can be built quite naturally. For a state of the model we take a vector of values of all program variables, which can be divided into two parts. The first part is a value vector of inputs at the moment of the beginning of a new PLC working cycle. The second part is a value vector of outputs and internal variables after passing a complete working cycle (on the inputs from the first part). In other words, the state of the model is a state of a PLC-program after the complete passing of a working cycle. Thus, a transition from one state to another depends on the (previous) values of the outputs and internal variables of the first state and the (new) values of the inputs of the second state. For each state, the degree of the transition relation branching is determined by the number of all possible combinations of PLC input signals. Atomic propositions of the model are logical expressions on PLC program variables with using arithmetic and relational operators.

III. PROGRAMMING CONCEPT

A purpose of the article is to describe an approach to programming PLC, which would provide a possibility of PLC-program correctness analysis by the model checking method. We will proceed from convenience and simplicity of using the model checking method. We require holding two following conditions.

Condition 1. The value of each variable must not change more than once per one full execution of the program while passing the PLC working cycle.

Condition 2. The value of each variable must change at only one place in the program in some operation block without nestings.

This conditions are reasonable assumption because inputs are always latched while operating the cycle. We will change the variable value only when it is really necessary, i. e. we will forbid an access to the variable by assigning if conditions of mandatory changing of its value do not hold. In this approach, the requirements of changing the value of a certain variable V after one pass of the PLC working cycle are represented by the following LTL formulas.

The next LTL formula is used for describing situations, leading to an increase of the variable value V

$$\mathbf{GX}(V > _V \Rightarrow OldValCond \wedge FiringCond \wedge V = NewValExpr)(1)$$

This formula means that whenever a new value of variable V is larger than its previous value, recorded in the variable $_V$, it follows that the old value of variable V satisfies the condition *OldValCond*, a condition of the external action *FiringCond* is accomplished, and the new value of variable V is the value of the expression *NewValExpr*.

The leading underscore symbol “ $_$ ” in the denotation of the variable $_V$ is taken as a pseudo-operator, allowing to refer to

the previous state value of a variable V . This pseudo-operator can be used only under the scope of the temporal operator \mathbf{X} .

Conditions $FiringCond$ and $OldValCond$ are logical expressions on program variables and constants, which are constructed using comparison operators, logical and arithmetic operators and the pseudo-operator “_”. By definition, the pseudo-operator can be applied only to variables. The expression $FiringCond$ describes situations, when changing the value of the variable V is needed (if it is allowed by the condition $OldValCond$). The expression $NewValExpr$ is built using variables and constants, comparison, logical and arithmetic operators and the pseudo-operator “_”.

For descriptions of all possible increasing value situations the formula (1) may have several sets of considered conjunctive parts $OldValCond_i \wedge FiringCond_i \wedge V = NewValExpr_i$, combined in a disjunction, after the operator \Rightarrow .

Situations that lead to a decrease of V value are described similarly:

$$\mathbf{GX}(V < _V \Rightarrow OldValCond' \wedge FiringCond' \wedge V = NewValExpr')(1')$$

Temporal formulas of the form (1) and (1') describe a desired behavior of some integer variable. A more simple LTL formula is proposed to use in case of a logical (binary) data type variable. The following formula describes situations which increase the value of a binary variable V :

$$\mathbf{GX}(_V \wedge V \Rightarrow FiringCond). \quad (2)$$

Situations that lead to a decrease of the variable V value are described similarly:

$$\mathbf{GX}(_V \wedge \neg V \Rightarrow FiringCond'). \quad (2')$$

Let's consider a special case of the specification form (1) and (1'), where for V we have

$$\begin{aligned} FiringCond &= FiringCond' = 1, \\ NewValExpr &= NewValExpr', \\ OldValCond &= (_V < NewValExpr) \text{ and} \\ OldValCond' &= (_V > NewValExpr): \end{aligned}$$

$$\mathbf{GX}(V > _V \Rightarrow _V < NewValExpr \wedge V = NewValExpr);$$

$$\mathbf{GX}(V < _V \Rightarrow _V > NewValExpr \wedge V = NewValExpr).$$

This specification can be replaced by the following LTL formula:

$$\mathbf{GX}(V = NewValExpr). \quad (3)$$

The variable V we will call a *register-variable*, if it has specification of forms (1), (1'), (2) and (2'). If V is constructed by specification of the form (3), it is called a *function-variable*. In the special case of specification (3), where the expression $NewValExpr$ does not contain leading underscore pseudo-operator “_”, variable V is called a *substitution-variable*.

It is important to note that each LTL formula template is constructive, i.e. the program can be easily build from specification that would correspond to temporal properties expressed by these formulas. Thus, we can say that PLC programming is reduced to building a behavior specification of each program variable, which is output or auxiliary internal

variable. The process of writing a program code is completed, when specification for each such variable is created. Note, that quantity and meaning of output variables are defined by a PLC and a problem formulation.

The program specification is divided into two parts: 1) specification of a behaviour of all program variables (except inputs), 2) specification of common program properties. The second part of specification affects quantity and meaning of internal auxiliary PLC program variables.

In specification it is important to consider the order of temporal formulas describing the behavior of the variables. A variable without the pseudo-operator “_” may be involved in the specification of another variable behavior only if the specification of its behavior is completed and is in the text above.

If necessary, we will use the keyword “Init” for indication of a variable initial value. For example, $Init(V) = 1$ means that the variable V initially is set to 1. If the initial value of some variable is not explicitly defined, it is assumed that this value is zero.

IV. PROGRAMMING BY SPECIFICATION

In this section, we consider a way of constructing a program code by constructive LTL-specification of the program variable behavior. In general, a translation process from LTL-formulas to program code is the following. Two temporal formulas of variable V , marked $V+$ (value increase, (1)) and $V-$ (value decrease, (1')), are set in conformity to the text block in the ST language:

```
IF    OldValCond AND FiringCond THEN
    V := NewValExpr;                (* V+ *)
ELSIF OldValCond' AND FiringCond' THEN
    V := NewValExpr';              (* V- *)
END_IF;
```

in the IL language:

```
calculation of OldValCond AND FiringCond
JMPCN VL1
calculation of NewValExpr
ST     V
JMP    VLEND

VL1:   calculation of OldValCond' AND FiringCond'
JMPCN VL2
calculation of NewValExpr'
ST     V
JMP    VLEND

VL2:
VLEND:
```

If the number of conjunctive blocks

$$OldValCond_i \wedge FiringCond_i \wedge V = NewValExpr_i$$

in LTL formulas will be more than two, the number of alternative branches “ELSIF” or labels will grow (by one branch or label for each new block).

Note, that the behavior of the obtained program will completely satisfy LTL formulas of specifications.

For LTL formula of a boolean variable V behavior in the forms $V+$ (2) and $V-$ (2'), we have the following ST-block:

```
IF NOT  $\_V$  AND  $FiringCond$  THEN  $V := 1$ ; (*  $V+$  *)
ELSIF  $\_V$  AND  $FiringCond'$  THEN  $V := 0$ ; (*  $V-$  *)
END_IF;
```

IL-block:

```
calculation of NOT  $\_V$  AND  $FiringCond$ 
JMPCN VL1
S      V
JMP    VLEND
```

VL1:

```
calculation of  $\_V$  AND  $FiringCond'$ 
JMPCN VL2
R      V
JMP    VLEND
```

VL2:

VLEND:

LD-block (in LD only boolean variable are used):

```
 $\_V$       V
-|/|----- $FiringCond$ ------(S)-----
 $\_V$       V
-| |----- $FiringCond'$ ------(R)-----
```

Each program variable must be defined in the description section (local or global) and initialized in conformity with the specification. Note that, for example, in CoDeSys [8] all variables are initialized to zero by default.

In addition, we must implement the idea of the pseudo-operator “_”. To do this, in the end of the program an area for a pseudo-operator section is allocated. In this area an assignment $_V := V$ is added after description of the behavior of all specification variables. In IL this assignment is:

```
LD      V
ST       $\_V$ 
```

An assignment in LD:

```
 $V$        $\_V$ 
-----|------( )-----
```

The assignment is added for each variable V , to the last value of which is addressed as $_V$. The variable $_V$ is also necessary to define in the description section with the same initialization as for the variable V .

Note, that the approach to programming by specification, which describes the reason of changing each program variable value, looks very natural and reasonable, because a PLC output signal is the control signal, and changing the value of this control signal usually carries an additional meaning. For example, it is important to understand why an engine or some lamp must be turned on/off. Therefore, it seems quite obvious that every variable must be accompanied by two properties, one for each direction changing. It is assumed that if change conditions are not made, the variable remains at its previous state.

V. BUILDING SMV-MODEL BY SPECIFICATION

We consider the verifier Cadence SMV [13] as a software tool of correctness analysis by model checking method. It is

proposed to build a Kripke structure model in the SMV language with further verification of common program properties satisfiability for this model after creating the specification. If some common program property is not hold for the model, the verifier builds an example of incorrect path in a Kripke structure model, by which corrections in the specification are produced. And only after all the program properties have been verified with positive results, ST-program of PLC is built by specification.

The SMV language allows to define a variable value in the next state of a model by using the “next” operator. Branching of the transition relation is provided by the “nondeterministic” assignment. For example, assignment $next(V) := \{0, 1\}$ means that states and transitions to them will be generated both with a value of $V = 0$, and with a value of $V = 1$. In the SMV language the symbols “&”, “|”, “~” and “->” denote logical “and”, “or”, “not” and implication, respectively.

The SMV language is oriented on creating the next states of Kripke models from the current state. The initial current state of the model is the state of program after initialization. Therefore, specification of the behavior of a variable V (1) and (1') will be easier (clearer) to rewrite in the following equivalent form

$$V+: \mathbf{G}(\mathbf{X}(V > _V) \Rightarrow \mathbf{X}(OldValCond) \wedge \mathbf{X}(FiringCond) \wedge \mathbf{X}(V = NewValExpr)),$$

$$V-: \mathbf{G}(\mathbf{X}(V < _V) \Rightarrow \mathbf{X}(OldValCond') \wedge \mathbf{X}(FiringCond') \wedge \mathbf{X}(V = NewValExpr')).$$

And then we get an SMV-model of a variable V behavior quite naturally, putting the “next” operator in conformity to the temporal operator \mathbf{X} :

```
case{ next(OldValCond) & next(FiringCond) :
      next(V) := next(NewValExpr);
      next(OldValCond') & next(FiringCond') :
      next(V) := next(NewValExpr');
default :
      next(V) := V; }.
```

Keyword “default” means what must happen by default, i. e. if conditions of first two branches in the “case” block don't hold.

In the case of a boolean variable V specification (2) and (2') is converted to the following SMV-model

```
case{ ~V & next(FiringCond) : next(V) := 1;
      V & next(FiringCond') : next(V) := 0;
default : next(V) := V; }.
```

A model of a function-variable behavior is defined as $next(V) := next(NewValExpr)$.

Let's now consider a specification of the behavior of a substitution-variable V . In this case $NewValExpr$ does not contain pseudo-operator “_”. This allows to rewrite the specification in the following equivalent form:

$$V: \mathbf{XG}(V = NewValExpr).$$

In fact, this formula means that if the initial state of the model does not considered, then an equality $V = NewValExpr$ must hold in all other states of the model. Fairness of formula $\mathbf{XG}(V = NewValExpr)$ follows from the fairness of more

general formula $G(V = NewValExpr)$. Therefore, more general formula can be used as the constructive specification for building SMV-model of a substitution-variable V . SMV-model is built by this specification in the form of assignment

$$V := NewValExpr.$$

The Cadence SMV verifier allows to check program models, containing up to 59 binary variables (all variables in SMV are represented by sets of binary variables). The substitution-variables are not included in this number, i.e. only register-variables and function-variables are considered.

VI. CONCLUSION

The approach has been successfully approved on some (about a dozen) “discrete” logical control problems of different types with the average number of binary PLC inputs and outputs about 30 and the total number of binary program variables up to 59. For example, properties, relating to maintenance of the technological process (to exclude the possibility of nonconforming product outflow), were verified for a PLC program, controlling a mixture preparation device. Properties relating to connection standby pumps in time were tested for problem of hydraulic pump system control. And properties of mandatory execution of received commands of cabin lift calling were tested for library lift control problem. The verification work was carried out on PC with processor Intel Core i7 2600K 3.40 GHz. Time spent by verifier Cadence SMV to check specified properties is limited to a few seconds.

The further work is to build software tools for modeling, specification, construction and verification of PLC-programs, according to the results of the work on this topic.

ACKNOWLEDGEMENTS

This work was financially supported by the Russian Foundation for Basic Research (project no. 12-01-00281-a).

REFERENCES

- [1] *Kuzmin E. V., Sokolov V. A.* Modeling, Specification and Construction of PLC-programs // Modeling and analysis of information systems. 2013. V. 20, No. 2. P. 104–120 [in Russian].
- [2] *Kuzmin E. V., Sokolov V. A.* On Construction and Verification of PLC-programs // Modeling and analysis of information systems. 2012. V. 19, No. 4. P. 25–36. [In Russian].
- [3] *Kuzmin E. V., Sokolov V. A.* On Verification of PLC-programs Written in the LD-Language // Modeling and analysis of information systems. 2012. V. 19, No. 2. P. 138–144. [In Russian].
- [4] *Kuzmin E. V., Sokolov V. A., Ryabukhin D. A.* Construction and Verification of PLC LD-programs by LTL-specification // Modeling and analysis of information systems. 2013. V. 20, No. 6. P. 78–94 [in Russian].
- [5] *Petrov I. V.* Programmiruemye kontrollery. Standartnye jazyki i priemy prikladnogo proektirovaniya. M.: SOLON-Press, 2004. 256 p. [In Russian].
- [6] *Canet G., Couffin S., Lesage J.-J., Petit A., Schnoebelen Ph.* Towards the Automatic Verification of PLC Programs Written in Instruction List // Proc. of the IEEE International Conference on Systems, Man and Cybernetics. Argos Press, 2000. P. 2449–2454.
- [7] *Clark E. M., Grumberg O., Peled D. A.* Model Checking. The MIT Press, 2001.
- [8] CoDeSys. Controller Development System. <http://www.3s-software.com/>
- [9] *Gries D.* The Science of Programming. Springer-Verlag, 1981.
- [10] *Parr E. A.* Programmable Controllers. An engineer’s guide. Newnes, 2003. 442 p.

- [11] *Pavlovic O., Pinger R., Kollman M.* Automation of Formal Verification of PLC Programs Written in IL // Proc. of 4th International Verification Workshop (VERIFY’07). Bremen, Germany, 2007. P. 152–163.
- [12] *Rossi O., Schnoebelen Ph.* Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs // Proc. of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems, Shaker Verlag, 2000. P. 177–182.
- [13] SMV (Symbolic Model Verifier). The Cadence SMV Model Checker. <http://www.kenmcil.com/smv.html>

APPENDIX A

Library lift

A library lift scheme is represented in Fig. 1. The purpose of the lift is to lift up books on request from the basement to the first and second floors of the library and to return them to the basement.

The elevator cabin is called from the base floor by pressing buttons “Up 2” and “Up 1”. If the corresponding command was accepted, this button lamp turns on. It turns off when the command is done. When the cabin is on some floor, the lamps “Floor 2”, “Floor 1” or “Floor 0” are on. There are shaft doors, which are opened and closed manually. The sensors “DS2”, “DS1” and “DS0” are needed to determine the position of doors. The door sensor is on if door is closed. The signal from the door sensor is determined using a lamp of the sensor.

The floor sensor “FS” in the elevator cabin is used for finding the position of the cabin in the shaft. The floor sensor is on, if the cabin is entirely on a particular floor. Otherwise, the signal is removed.

The library lift control is carried out using a PLC receiving input signals from sensors and buttons and sending output signals to the lift motor and lamps. The task is to construct a PLC program with 10 binary inputs and 14 binary outputs for controlling the lift. PLC interface is shown in Fig. 2. More detailed requirements for the library lift program are given in the article [4]. The constructive specification of the library lift control program was built according to these requirements and the programming concept by LTL-specification.

```

Ctr0+: GX(~_Ctr0 & Ctr0 -> FS & ~_FS &
        _Ctr1 & ~_Dir);
Ctr0-: GX(_Ctr0 & ~Ctr0 -> FS & ~_FS);
Ctr1+: GX(~_Ctr1 & Ctr1 -> FS & ~_FS &
        (_Ctr2 & ~_Dir | _Ctr0 & _Dir));
Ctr1-: GX(_Ctr1 & ~Ctr1 -> FS & ~_FS);
init(Ctr1)=1;
Ctr2+: GX(~_Ctr2 & Ctr2 -> FS & ~_FS &
        _Ctr1 & _Dir);
Ctr2-: GX(_Ctr2 & ~Ctr2 -> FS & ~_FS);
Up01+: GX(~_Up01 & Up01 -> FS & Ctr0 & PBU01);
Up01-: GX(_Up01 & ~Up01 -> FS & Ctr1 & ~_Mtr);
Up02+: GX(~_Up02 & Up02 -> FS & Ctr0 & PBU02);
Up02-: GX(_Up02 & ~Up02 -> FS & Ctr2 & ~_Mtr);
Dwn2+: GX(~_Dwn2 & Dwn2 -> FS & Ctr2 & ~_Mtr &
        (PBDwn2 | _Tmr.Q));
Dwn2-: GX(_Dwn2 & ~Dwn2 -> FS & Ctr0 & ~_Mtr);
Dwn1+: GX(~_Dwn1 & Dwn1 -> FS & Ctr1 & ~_Mtr &
        (PBDwn1 | _Tmr.Q));
Dwn1-: GX(_Dwn1 & ~Dwn1 -> FS & Ctr0 & ~_Mtr);
init(Dwn1)=1;
Flr2+: GX(~_Flr2 & Flr2 -> PBF1r2);
Flr1+: GX(~_Flr1 & Flr1 -> PBF1r1);
Flr2-: GX(_Flr2 & ~Flr2 -> FS & Ctr2 & ~_Mtr);

```



```

JMPCN Up01LEND
R      Up01
JMP   Up01LEND

```

```

Up01L2:
Up01LEND:

```

ST-program is built by constructive LTL-specification after checking common program properties.

SMV-model of the “library lift” program

Modeling of a timer is described in [1]. A model of behaviour of floor sensor FS is following. When the motor is on, a value of the sensor FS can be changed. When the motor is off, a value of the sensor FS remains unchanged. Fairness conditions for FS mean that when the motor is on, floor sensor can not remain unchanged indefinitely.

```

module timer(){
  I : 0..1; /* Input */
  Q : 0..1; /* Output */
  init(I):=0; init(Q):=0;
  next(Q):= next(I) & (Q | {0, 1});
  FAIRNESS I -> Q;
}
module main(){ /* Inputs */
/* Buttons */
  PBFlr2, PBDwn2, PBFlr1, PBDwn1, PBU02, PBU01: 0..1;
/* Sensors */
  DS2, DS1, DS0, FS: 0..1;
/* Outputs */
  Mtr, Dir: 0..1; /* Motor */
  Flr2Lmp, Flr1Lmp, Flr0Lmp, DS2Lmp, DS1Lmp,
  DS0Lmp: 0..1; /* Lamps */
/* Commands */
  Flr2, Flr1, Dwn2, Dwn1, Up02, Up01: 0..1;
/* Auxiliary */
  Ctr0, Ctr1, Ctr2: 0..1;
  DS: 0..1;
  Tmr : timer;
/* Initialization section */
/* Inputs */
  init(PBFlr2):=0; init(PBFlr1):=0;
  init(PBU02):=0; init(PBU01):=0;
  init(PBDwn2):=0; init(PBDwn1):=0;
  init(DS1):=0; init(DS0):=0;
  init(DS2):=0; init(FS):=0;
/* Outputs */
  init(Mtr):=0; init(Flr1):=0; init(Dwn1):=1;
  init(Up01):=0; init(Dir):=0; init(Flr2):=0;
  init(Dwn2):=0; init(Up02):=0;
/* Auxiliary */
  init(Ctr0):=0; init(Ctr1):=1; init(Ctr2):=0;
/* Transition system */
/* Inputs */
  next(PBFlr2)={0, 1}; next(PBDwn2)={0, 1};
  next(PBFlr1)={0, 1}; next(PBDwn1)={0, 1};
  next(PBU02)={0, 1}; next(PBU01)={0, 1};
  next(DS2)={0, 1}; next(DS1)={0, 1};
  next(DS0)={0, 1};
  case{ Mtr : next(FS)={0,1};
        default : next(FS)=FS;};
  FAIRNESS Mtr -> FS;
  FAIRNESS Mtr -> ~FS;
/* Outputs and auxiliary */
  case{~Ctr0 & next(FS) & ~FS & Ctr1 & ~Dir :
    next(Ctr0)=1; /* Ctr0+ */
    Ctr0 & next(FS) & ~FS : /* Ctr0- */
    next(Ctr0)=0;
    default : next(Ctr0)=Ctr0; };
  case{~Ctr1 & next(FS) & ~FS &
  (Ctr2 & ~Dir | Ctr0 & Dir) :
    next(Ctr1)=1; /* Ctr1+ */

```

```

  Ctr1 & next(FS) & ~FS :
    next(Ctr1)=0; /* Ctr1- */
    default : next(Ctr1)=Ctr1;};
  case{~Ctr2 & next(FS) & ~FS & Ctr1 & Dir:
    next(Ctr2)=1; /* Ctr2+ */
    Ctr2 & next(FS) & ~FS : /* Ctr2- */
    next(Ctr2)=0;
    default : next(Ctr2)=Ctr2;};

  case{~Up01 & next(FS) & next(Ctr0) & next(PBU01):
    next(Up01)=1; /* Up01+ */
    Up01 & next(FS) & next(Ctr1) & ~Mtr:
    next(Up01)=0; /* Up01- */
    default : next(Up01)=Up01;};
  case{~Up02 & next(FS) & next(Ctr0) & next(PBU02):
    next(Up02)=1; /* Up02+ */
    Up02 & next(FS) & next(Ctr2) & ~Mtr:
    next(Up02)=0; /* Up02- */
    default : next(Up02)=Up02;};
  case{~Dwn2 & next(FS) & next(Ctr2) &
  ~Mtr & (next(PBDwn2) | Tmr.Q):
    next(Dwn2)=1; /* Dwn2+ */
    Dwn2 & next(FS) & next(Ctr0) & ~Mtr:
    next(Dwn2)=0; /* Dwn2- */
    default : next(Dwn2)=Dwn2;};
  case{~Dwn1 & next(FS) & next(Ctr1) &
  ~Mtr & (next(PBDwn1) | Tmr.Q):
    next(Dwn1)=1; /* Dwn1+ */
    Dwn1 & next(FS) & next(Ctr0) & ~Mtr:
    next(Dwn1)=0; /* Dwn1- */
    default : next(Dwn1)=Dwn1;};
  case{~Flr2 & next(PBFlr2):
    next(Flr2)=1; /* Flr2+ */
    Flr2 & next(FS) & next(Ctr2) & ~Mtr:
    next(Flr2)=0; /* Flr2- */
    default : next(Flr2)=Flr2;};
  case{~Flr1 & next(PBFlr1):
    next(Flr1)=1; /* Flr1+ */
    Flr1 & next(FS) & next(Ctr1) & ~Mtr:
    next(Flr1)=0; /* Flr1- */
    default : next(Flr1)=Flr1;};
  case{~Dir & next(FS) & next(Ctr0) & ~Mtr:
    next(Dir)=1; /* Dir+ */
    Dir & next(FS) & (next(Ctr2) | next(Ctr1) &
    next(Dwn1) & ~next(Up02) & (~next(Flr2) |
    next(Dwn2))) & ~Mtr:
    next(Dir)=0; /* Dir- */
    default : next(Dir)=Dir;};
  DS:= DS0 & DS1 & DS2; /* DS */
  case{~Mtr & next(DS) &
  (~next(FS) & (next(Dwn1)|next(Dwn2)|next(Up01)|
  next(Up02)|next(Flr1)|next(Flr2))|
  next(FS) & next(Ctr0) & (next(Up01)|next(Up02)|
  Tmr.Q & (next(Flr1) | next(Flr2))) |
  next(FS) & next(Ctr1) & next(Dwn1) |
  next(FS) & next(Ctr2) & next(Dwn2)):
    next(Mtr)=1; /* Mtr+ */
    Mtr & (~next(DS) | next(FS) & next(Ctr0) & ~Dir |
    next(FS) & next(Ctr2) & Dir | next(FS) &
    next(Ctr1) & (next(Flr1) & ~next(Dwn1)|next(Up01))) :
    next(Mtr)=0; /* Mtr- */
    default : next(Mtr)=Mtr;};
  next(Tmr.I)= next(~Mtr & FS &
  (Ctr0 & DS0 | Ctr1 & DS1 | Ctr2 & DS2));
  Flr0Lmp:= ~Mtr & FS & Ctr0; /* Flr0Lmp */
  Flr1Lmp:= ~Mtr & FS & Ctr1; /* Flr1Lmp */
  Flr2Lmp:= ~Mtr & FS & Ctr2; /* Flr2Lmp */
  /* DS0Lmp, DS1Lmp, DS2Lmp */
  DS0Lmp:= DS0; DS1Lmp:= DS1; DS2Lmp:= DS2;
/* Properties section */
  P_Ctr: assert G(Ctr0+Ctr1+Ctr2 = 1);
  P_Limit0: assert G~(FS & Ctr0 & Dir=0 & Mtr);
  P_Limit2: assert G~(FS & Ctr2 & Dir=1 & Mtr);
  P_Doors: assert G(Mtr -> DS);

```

```

P_Stop: assert G(~Mtr & DS & ~FS);
P_Mtr: assert G( Mtr -> F(~Mtr));
P_Flr2: assert G(Flr2 -> ~G(Flr2 U ~DS)) ->
  G(Flr2 -> F(~Flr2));
P_Flr1: assert G(Flr1 -> ~G(Flr1 U ~DS)) ->
  G(Flr1 -> F(~Flr1));
P_Up01: assert G(Up01 -> ~G(Up01 U ~DS)) ->
  G(Up01 -> F(~Up01));

P_Up02: assert G(Up02 -> ~G(Up02 U ~DS)) ->
  G(Up02 -> F(~Up02));
P_Dwn1: assert G(Dwn1 -> ~G(Dwn1 U ~DS)) ->
  G(Dwn1 -> F(~Dwn1));
P_Dwn2: assert G(Dwn2 -> ~G(Dwn2 U ~DS)) ->
  G(Dwn2 -> F(~Dwn2));
P_Move: assert G(DS & ~FS -> Mtr);
}

```

ST-program of "library lift"

```

VAR_GLOBAL
(* Inputs *)
PBFlr2,PBDwn2,PBFlr1, PBDwn1, PBU02,PBU01: BOOL;
DS2, DS1, DS0, FS: BOOL;
(* Outputs *)
Mtr, Dir, Flr2Lmp, Flr1Lmp, Flr0Lmp, DS2Lmp: BOOL;
DS1Lmp, DS0Lmp, Flr2, Flr1, Dwn2, Up02, Up01: BOOL;
Dwn1: BOOL :=1;
END_VAR

PROGRAM PLC_PRG
VAR
  Tmr: TON := (PT := T#10s);
  Ctr0, Ctr2: BOOL;
  Ctr1: BOOL := TRUE;
  _Ctr, _FS, _Dir, _Dwn1, _Dwn2, _Flr1, _Flr2: BOOL;
  _Mtr, _TmrQ, _Up01, _Up02: BOOL;
END_VAR
IF NOT _Ctr0 AND FS AND NOT _FS AND
  _Ctr1 AND NOT _Dir THEN
  Ctr0:=1; (* Ctr0+ *)
ELSIF _Ctr0 AND FS AND NOT _FS THEN
  Ctr0:=0; (* Ctr0- *)
END_IF;
IF NOT _Ctr1 AND FS AND NOT _FS AND
  (_Ctr2 AND NOT _Dir OR
  _Ctr0 AND _Dir) THEN
  Ctr1:=1; (* Ctr1+ *)
ELSIF _Ctr1 AND FS AND NOT _FS THEN
  Ctr1:=0; (* Ctr1- *)
END_IF;
IF NOT _Ctr2 AND FS AND NOT _FS AND
  _Ctr1 AND _Dir THEN
  Ctr2:=1; (* Ctr2+ *)
ELSIF _Ctr2 AND FS AND NOT _FS THEN
  Ctr2:=0; (* Ctr2- *)
END_IF;
IF NOT _Up01 AND FS AND Ctr0 AND PBU01 THEN
  Up01:=1; (* Up01+ *)
ELSIF _Up01 AND FS AND Ctr1 AND NOT _Mtr THEN
  Up01:=0; (* Up01- *)
END_IF;
IF NOT _Up02 AND FS AND Ctr0 AND PBU02 THEN
  Up02:=1; (* Up02+ *)
ELSIF _Up02 AND FS AND Ctr2 AND NOT _Mtr THEN
  Up02:=0; (* Up02- *)
END_IF;
IF NOT _Dwn2 AND NOT _Mtr AND FS AND Ctr2 AND
  (PBDwn2 OR _TmrQ) THEN Dwn2:=1; (* Dwn2+ *)
ELSIF _Dwn2 AND NOT _Mtr AND FS AND Ctr0 THEN
  Dwn2:=0; (* Dwn2- *)
END_IF;
IF NOT _Dwn1 AND NOT _Mtr AND FS AND Ctr1 AND

```

```

  (PBDwn1 OR _TmrQ) THEN Dwn1:=1; (* Dwn1+ *)
ELSIF _Dwn1 AND NOT _Mtr AND FS AND Ctr0 THEN
  Dwn1:=0; (* Dwn1- *)
END_IF;
IF NOT _Flr2 AND PBFlr2 THEN Flr2:=1; (* Flr2+ *)
ELSIF _Flr2 AND NOT _Mtr AND FS AND Ctr2 THEN
  Flr2:=0; (* Flr2- *)
END_IF;
IF NOT _Flr1 AND PBFlr1 THEN Flr1:=1; (* Flr1+ *)
ELSIF _Flr1 AND NOT _Mtr AND FS AND Ctr1 THEN
  Flr1:=0; (* Flr1- *)
END_IF;
IF NOT _Dir AND NOT _Mtr AND FS AND Ctr0 THEN
  Dir:=1; (* Dir+ *)
ELSIF _Dir AND NOT _Mtr AND FS AND
  (Ctr2 OR Ctr1 AND Dwn1 AND NOT Up02 AND
  (NOT Flr2 OR Dwn2)) THEN Dir:=0; (* Dir- *)
END_IF;
DS:=DS0 AND DS1 AND DS2;

IF NOT _Mtr AND DS AND
  (NOT FS AND (Dwn1 OR Dwn2 OR
  Up01 OR Up02 OR Flr1 OR Flr2) OR
  FS AND Ctr0 AND
  (Up01 OR Up02 OR _TmrQ AND
  (Flr1 OR Flr2)) OR
  FS AND Ctr1 AND Dwn1 OR
  FS AND Ctr2 AND Dwn2) THEN Mtr:=1;(* Mtr+ *)
ELSIF _Mtr AND (NOT DS OR
  FS AND Ctr0 AND _Dir=0 OR
  FS AND Ctr2 AND _Dir=1 OR
  FS AND Ctr1 AND
  (Flr1 AND NOT Dwn1 OR Up01))
  THEN Mtr:=0; (* Mtr- *)
END_IF;
Tmr.In:= NOT Mtr AND FS AND
  (Ctr0 AND DS0 OR Ctr1 AND DS1 OR Ctr2 AND DS2);
Tmr();
Flr0Lmp:= NOT Mtr AND FS AND Ctr0;
Flr1Lmp:= NOT Mtr AND FS AND Ctr1;
Flr2Lmp:= NOT Mtr AND FS AND Ctr2;
DS0Lmp:=DS0;
DS1Lmp:=DS1;
DS2Lmp:=DS2;
(* ----- pseudo-operator section -----*)
_TmrQ:=Tmr.Q; _FS:=FS; _Mtr:=Mtr;
_Dir:=Dir; _Ctr:=Ctr;
_Dwn1:=Dwn1; _Dwn2:=Dwn2;
_Flr1:=Flr1; _Flr2:=Flr2;
_Up01:=Up01; _Up02:=Up02;

```

An approach to lightweight static data race detection

Andrianov Pavel

Institute for System Programming
Russian Academy of Sciences,
Email: andrianov@ispras.ru

Khoroshilov Alexey

Institute for System Programming
Russian Academy of Sciences,
Email: khoroshilov@ispras.ru

Mutilin Vadim

Institute for System Programming
Russian Academy of Sciences,
Email: mutilin@ispras.ru

Abstract—The paper presents a lightweight approach to static data race detection. It is based on the Lockset one, but it implements several simplifications that are aimed to reduce amount of false alarms. The approach is implemented on top of CPAchecker tool and its evaluation is in progress. The main target of our research and evaluation is operating system kernels but the approach can be applied to analysis of other programs as well.

Keywords. static analysis, data race condition, verification, operating system kernel, shared data

I. INTRODUCTION

Despite a great progress in the field of software verification, errors associated with multithreading execution remain among the most difficult to identify. In addition concurrency bugs are rather numerous and, for example, make up about 20% of all bugs on average across file systems [1]. The most common causes of errors associated with the parallel execution of system code are data race conditions in which simultaneous access to shared data from multiple threads takes place. In particular the analysis of bug fixes for a year of Linux kernel development has shown that errors associated with data races are the most numerous class and make up 17% of typical errors [2].

At the moment there are two ways for finding data races automatically: dynamic analysis and static analysis. Dynamic analysis techniques allow to obtain a relatively small percentage of false alarms, but they are able to find data races only at those paths that occurred during the actual execution of a program. As far as data race requires two almost simultaneous accesses to the same data, this fact reduces chance of its detection. Also it is known that a significant number of execution paths are difficult to reproduce in test environment. Examples of tools, implemented such method, are Eraser [3], RaceHound [4] and DataCollider [5].

Methods of static analysis have the other problems. The lightweight methods, e.g. method, implemented in the tool Locksmith [6], analyze source code superficially. Such methods allow to find simple errors and work very fast. But the number of false alarms is about 90% on device drivers and about 98% on some POSIX applications [7]. The idea of such methods can be improved to decrease number of false alarms. The another way is heavyweight analysis. It is much more precise, but requires a lot of time. In case of data race detection total number of places, where data race can occur, is too large. There are some experiments with verification of kernel modules source code, for example, DDVerify [8]. But the results showed that the heavyweight multithreaded

analysis does not scale on such code. They got a combinatorial explosion of states, so, even for small modules the amount of required time and memory was huge.

In operating system kernels parallelism is more complex and less precise, because many kernel functions can be executed in parallel and it is difficult to define when the parallel execution can start. So finding data races in operating system kernel is much more difficult than in user-level programs.

So there is a need to create a lightweight method of static analysis which is easy to scale to large amounts of source code and allows to find most of error cases while keeps false alarms rate at reasonable level.

In this paper we suggest a new method of static analysis for data race detection, implemented on top of CPAchecker tool [9].

The rest of the paper is organized as follows. In the Section 2 required definitions are given. After that the idea of analysis is presented. The Section 4 describes the idea of configurable program analysis. After that implementation of our method is given. Then a need of annotations of source code is explained. The Section 7 talks about the solution architecture. After that the visualization of results is presented. In conclusion we briefly speak about the results, related work and future plans.

II. DEFINITIONS

In this article term thread is used to represent independent thread of execution in operating system kernel, for example, interrupt handlers and system calls executed on behalf of user space threads.

A **lock** is an object used for concurrent memory access exclusion. If lock is acquired from one thread the another thread trying to acquire the same lock can not continue its execution before the lock is released. For example, mutexes and spin locks are typical examples of such locks. We consider the kernel specific synchronization mechanisms such as disabling of interrupts and scheduling as specific locks as well. The lock can be linked with an address. For example, function `mutex_lock(&mutex)` linked with the address `&mutex`.

Shared data — an area of memory which is available from several threads. In C language shared data is presented by global variables and pointers to memory which is accessible from several threads via legal C constructions. It is important to note that sharedness is a characteristic of time. A local data can become shared at one point and return its local status later.

Usage of data — read or write data access.

Data race condition is a situation when there are two concurrent access to the same shared data and at least one of the access events is writing. Data race does not always lead to an error (for example, access to statistics counter), but it is a symptom of it.

III. LIGHTWEIGHT METHOD OF DATA RACE DETECTION

Our method is based on algorithm Lockset [3]. This algorithm considers two usages of the same data as a data race if these usages occur with disjoint sets of locks. The algorithm Lockset stores locks for every thread and set C of potential candidates of locks for every usage of shared data. If a usage occurs it intersects C and set of acquired locks for current thread and obtain new set \tilde{C} . If the last one is empty this is potential data race.

As far as our goal is to develop a lightweight static analysis method to analyze large amount of source code we apply some simplifications.

Our first heuristics is a definition if shared data is the same. We do not analyze a memory model but consider memory locations as the same only by syntax rules. For variables the equality of memory locations follows only from the equality of variables. Pointers with equal names are always considered to be pointed to the same memory area.

The second simplification makes a try to decrease a number of false alarms. We generate warnings only for usages of data, protected by locks at least once. If there are two usages without locks at all we do not generate warning. This is rather strong limitation because many real errors can occur exactly without locks.

The last simplification is the model of threads. We consider that every kernel API function, specified in documentation, could be executed in parallel with other one, including itself. The actual interrelation between kernel API functions is more difficult.

IV. CONFIGURABLE STATIC ANALYSIS

As far as our method implementation is based on Configurable Program Analysis (CPA) [9] let us briefly describe it.

The main idea of configurable static analysis is to combine advantages of data flow analysis and model checking methods and create a way to configure its interrelation. On the one hand, in data flow analysis algorithm works with control-flow graph (CFG) of program, disseminating information along the edges, on the other hand, model checking algorithms of heavyweight analysis unroll the tree of reachable states until one of the states will not be covered by another state.

In configurable static analysis proposed in [9] it is allowed to configure the analysis algorithm CPA choosing the merging operator and way to check the completion of the analysis. In addition, configurable static analysis can be configured of several algorithms CPA offering different types of analysis.

Configurable program analysis $(D, transfer, merge, stop)$ consists of an abstract domain D , transfer relation $transfer$, merge operator $merge$, the stop operator $stop$ as described below. These four components configure

analysis algorithm and affect the accuracy and the resources required for analysis.

Abstract domain D specifies the set of abstract states. Every abstract state is matched to its abstract value, i.e. set of concrete states, which it represents. Concrete state of program is a mapping of program variables set into the values of these variables.

Transfer relation $transfer$ determines for each abstract state e potential following abstract states $\{e'\}$, where each transition is marked by an edge of the CFG.

Operator $merge$ allows to combine information from several paths of analysis. It determines, when two nodes of the reachability tree are merged into one and when they are analyzed individually. In the data flow analysis merging always happens, when nodes refer to the same point in the program. In classical methods of model checking nodes are never merged.

Operator $stop$ checks whether the current state is covered by set of given states (already completed state). It determines when consideration of a path is stopped in current node. In the data flow analysis stop occurs when there is no abstract state including new concrete states, i.e. a fixed point is reached. In the methods of model checking stop occurs when one set of concrete states corresponding an abstract state is a subset of states corresponding to some other abstract state.

Let us look at one example of CPAs configuration tree (Fig. 1).

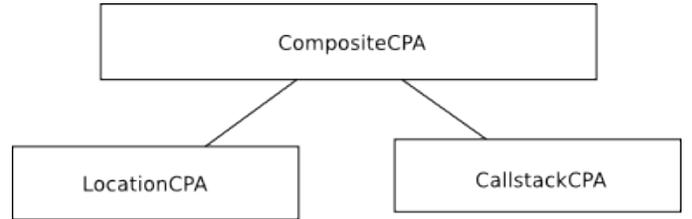


Fig. 1. Simple CPA configuration tree

We have three CPAs. The main is *CompositeCPA*. It includes *LocationCPA* and *CallstackCPA*.

State of *LocationCPA* contains only a line of source code. Hence its abstract domain is the set of possible line numbers.

Transfer relation changes the line number of current state to the line number of the edge successor.

Merge operator never merges states. *Stop* occurs only if we have already analyzed the current state before.

State of *CallstackCPA* consists of function calls stack. If we call a new function we push its name at top of the stack. If we return back we pull its name from the stack. It is the work of transfer relation. *Stop* and *merge* are the same as previous ones.

The aim of *CompositeCPA* is to combine CPAs mentioned above. Its domain is a cartesian product of *LocationCPA* domain and *CallstackCPA* domains. Transfer relation of *CallstackCPA* calls the containing transfer relations. First, it obtains a new state of *LocationCPA*, then a new state of *CallstackCPA*, and combines them together, thus we get the new state of *CompositeCPA*.

Merge and *stop* operators are also a combination of containing ones. To merge two states of *CompositeCPA*, first, Location states are merged, then Callstack ones are merged and finally they are combined into a next Composite state. The *stop* operator works in the similar way: if any containing CPA consider to stop analysis the analysis is stopped.

Consider, how the composition of CPAs analyzes the simple code:

```

1. int g(int a) {
2.     int b = 0;
3.     if (a == 0) {
4.         b++;
5.     }
6.     return b;
7. }
8. int f() {
9.     return 0;
10. }
11. int main() {
12.     int t;
13.     t = f();
14.     g(t);
15. }

```

In figure 2 there is a path of an analysis of the program above.

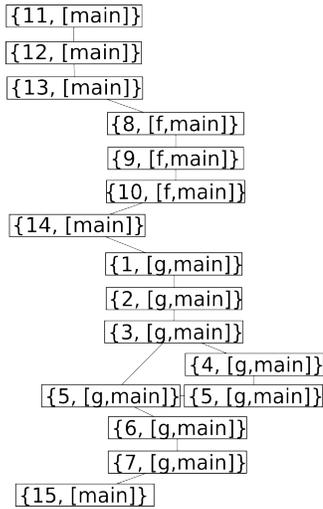


Fig. 2. Analysis path

First number in the braces is representing the state of *LocationCPA* (the line number) and after that follows the call stack of functions. Analysis starts from the main function, then it analyses the function *f*, then goes to *g*. In this function it meets if condition at line 3. It analyses two branches and gets the same resulting states at line 5. It means that one state is covered by another, so it continues the analysis with the only state.

V. IMPLEMENTATION

The implementation of the method is proposed to be into two stages. First of all, the shared data are identified, then for every usage of shared data the set of acquired locks is obtained.

Figure 3 represents these stages in CPAchecker-Lockator. The

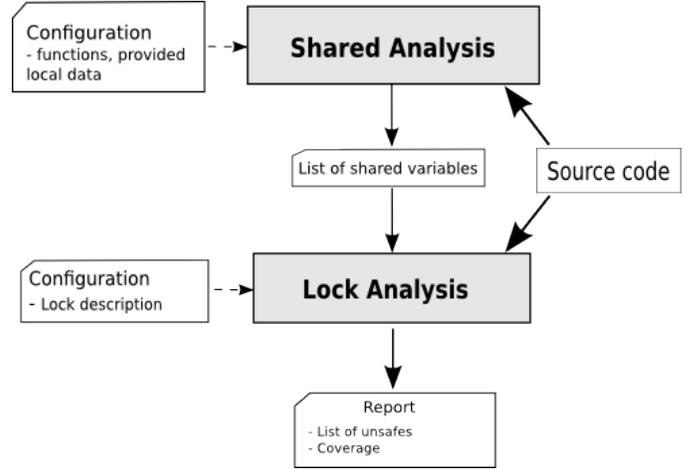


Fig. 3. Stages of analysis in CPAchecker-Lockator

configuration for *Shared analysis* consists of functions which introduce local data, for example, `calloc()`, `malloc()` and so on. We are sure that pointer returned by these functions points to local data and in current point of program it can not be shared. The configuration for *Lock analysis* includes locks descriptions and annotations which are described in section VI.

A. CPA configuration for Shared analysis

Shared analysis is used for collecting the list of shared variables in every point of a program, see Fig. 4.

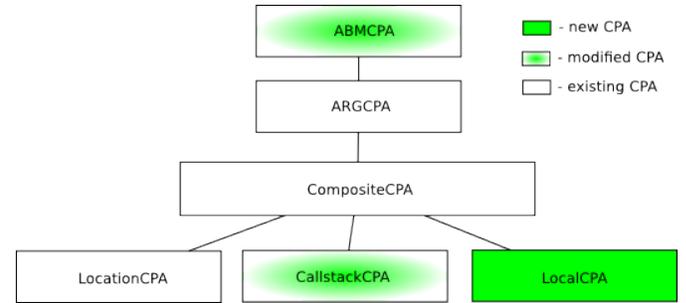


Fig. 4. Shared analysis configuration

BAMCPA (Block Abstraction Memorization) [10] — is responsible for modularity of analysis. If a function has been already analyzed with some state before the call and a set of resulting states on return from the function were already stored, the reanalysis of this function does not occur, the stored states are used instead.

ARGCPA (Abstract Reachability Graph) — is responsible for restoration of a path from current state to initial one. It stores parents and children for every state, so it can traverse all reached states and reestablish the path.

CompositeCPA — provides the analysis where the state is a product of the containing CPAs and the transfer is performed for all containing CPAs simultaneously.

LocationCPA — stores current line of source code in its state and is responsible for traversal of CFG.

CallstackCPA — stores a stack of called functions, so it is responsible for transferring by function calls and returns.

LocalCPA — is responsible for detecting locality of all variables accessible in current point of program. The data status can be *local*, *global* or *unknown*. The task of transfer operator is to spread the status variables for assignment operators and function calls. For example, if there is assignment $a = b$ then status of variable b is transferred to the variable a . At merge points the analysis joins results on the branches, such that for each variable the maximal status is taken. Considering the following example:

```
if (condition) {
  a = b;
} else {
  a = c;
}
```

At the merge point of the two branches of the if statement the status for a variable is taken as maximal status between then-branch and else-branch. Therefore, if b is *local* and c is *global*, then the result for a is *global*.

The result of this stage is a list of shared data. If we do not exactly know the sharedness we include the data into the list, thus considering it as shared.

B. CPA configuration for Lock analysis

Lock analysis is used for collecting a set of acquired locks for every usage of shared data, provided by previous stage of analysis.

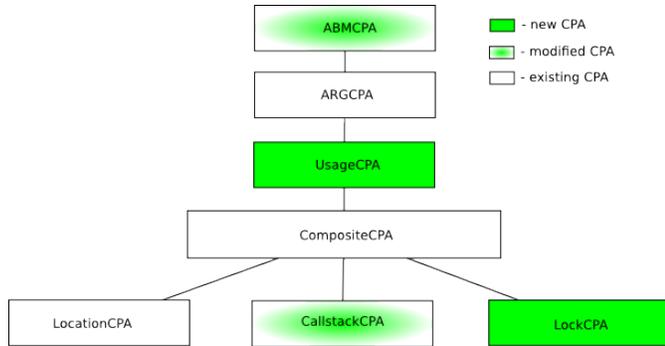


Fig. 5. Lock analysis configuration

ABMCPA, *ARGCPA*, *LocationCPA* and *CallstackCPA* are the same.

UsageCPA collects statistics of data usage. Transfer relation of *UsageCPA* identifies variables used in the expressions for read/write access and keeps the call stack for the usage, as well as a set of acquired locks.

At the end of analysis we obtain statistics about all usages for every shared data. The usage consists of:

- Set of acquired locks;
- Stack of function calls;
- The line number;
- CFG edge type (a function call expression, etc.);

- The type of access (READ, WRITE).

The *UsageCPA* is also used to establishing equality of variables, so they can be regarded as the same data for the analysis. This is required, for example, for lists, where the elements of the list usually have equal variable names like *next* given by the field name of the list structure. If we do not distinguish elements of different lists we get many false alarms, because the usages of different lists may be protected by different lock sets. That is why we want to bind the variable representing the elements to the list variable name to distinguish between the other lists. For this purpose the configurations contains functions which are used to work with the list. For example, the expression $e = \text{getElement}(\text{list})$ binds the variable e to the variable *list* passed as a parameter. In transfer relation upon detection of an annotated function the binding relation is changed accordingly.

LockCPA analyzes the set of acquired locks. Its state holds a set of locks acquired during the program execution. Each lock contains information about:

- Name of the lock;
- Recursive counter of acquires;
- Stack of function calls for every acquire.

Transfer relation changes the state of a plurality of acquired locks. When a lock acquisition function is called, the corresponding lock is added to the lock set or the counter is incremented. When a releasing function is called, the counter is decremented and if it becomes zero the lock is removed from the set.

States of all CPAs are never merged. Analysis stops if a state has been already analyzed.

VI. ANNOTATIONS

Let us consider the following chunk of code:

```
if (!isLOCKED) {
  lock();
}
global_var++;
if (!isLOCKED) {
  unlock();
}
```

In this example the increase of the global counter always occurs under the lock. Either someone acquires it earlier, or this function acquires it and releases it after increasing counter. But the analysis considers four paths because it can take if or else branch in both if statements. Two of these paths are infeasible, because the conditions in the if statements are the same. So at the end the analysis has two states of acquired lock sets: $\{\text{lock}\}$ and $\{\emptyset\}$, where the first one is not reachable in the real execution.

Such situations do not often occur, but each of them offers a significant number of false alarms, since the output of the function under lock affects all further paths of analysis. The annotation of functions are used to deal with such cases. It is a way to tell analysis that a function is always releases or acquires the lock.

Annotation describes function in terms of *LockCPA* states. After the function has been analyzed, the state is adjusted in accordance with the annotation.

Currently 4 types of specifications are supported:

- Acquiring a lock — function always acquires a lock.
- Releasing lock — function always releases a lock.
- Reseting a lock — if the lock can be acquired several times recursively, the function totally releases it.
- Restoring a lock — function does not modify set of locks, all changes should be forgotten.

VII. SOLUTION ARCHITECTURE

For race detection we reuse the Linux Driver Verification (LDV [11]) architecture developed within ISPRAS project for the verification of Linux operating system device drivers (see Fig. 6).

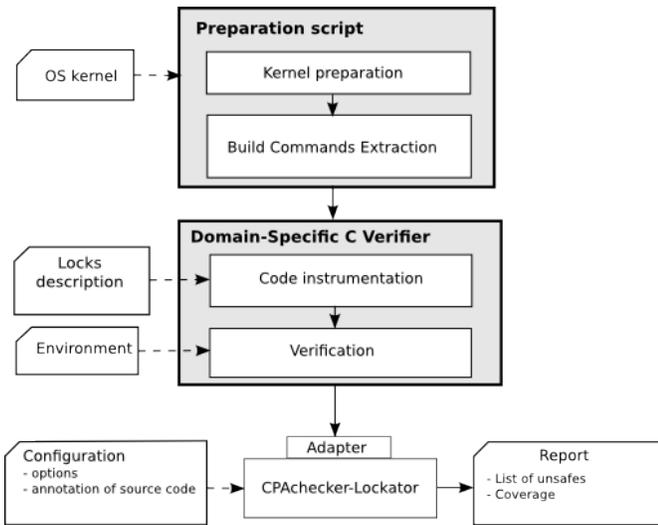


Fig. 6. Solution architecture

First kernel of operating system is prepared. During this stage compiler calls are replaced by our command extractor calls. Also other modules are prepared for building at this stage. Then build command stream is extracted by special scripts. Obtained command stream is transmitted to *Domain-Specific C Verifier* component. It instruments source code, using locks description. For example, it replaces macros used for acquisition and releasing locks by model functions, annotated in the configuration, because macros can be expanded to very difficult command sequence, while model function is easier to analyze.

Then model of environment is included. It is presented by *main* function containing system calls which can be executed in parallel according to documentation and interrupt handlers calls. So we consider that all functions called from *main* are executed simultaneously.

After all preparations the source code is analyzed by CPAchecker-Lockator. It generates report containing a list of unsafe cases with detail information about each of them.

Statistics	General	Unsafe
Global variables:	195	29
Simple:	122	23
Pointer:	73	6
Local variables:	3	0
Simple:	0	0
Pointer:	3	0
Structure fields:	118	24
Simple:	105	24
Pointer:	13	0
Total:	316	53

TABLE I. Example of general report for launch on Linux driver `floppy.ko`

VIII. VISUALIZATION OF RESULTS

To visualize the potential cases of data races another component of LDV Tools called *Error Trace Visualizer* is reused. When the tool generates a warning about the data race, it must be shown to the user. Moreover, the user should check if it is a false alarm or true error. Therefore it is necessary to present visualization of the unsafe error trace and its association with the context - source code under analysis. *Error Trace Visualizer* interprets the data received from the verifier, converts them and associates it with the source code. To represent the results the HTML-report is generated. The main page of a report contains general statistics (Tab. I). There are total numbers of variables of each of three categories: global, local and structure fields and number of variables, producing unsafes. The pointer variable means the access by pointer and simple one — the access to variable itself. Also the report lists all found locks. After that there is a list of all unsafes, that could potentially be a data race. For each unsafe the report contains a pair of usages with disjoint sets of locks.

Also there is an option to generate source code coverage. It shows the code which has been analyzed by the verifier and its relation to the whole kernel code.

An example of source code presentation is shown in Fig. 7.

Here we can see two functions called from the entry point (*main* function). The function `print` prints information about global variable, and `increase` increments its value with lock protection. There is a data race, because function `increase` can write to the variable simultaneously with the check in the function `print`. So, as a consequence of the race the printed output message may be wrong.

Our tool generates a warning for variable `global` with error trace shown in Fig. 8.

On the left side we can see the error call stack with points of acquiring locks and points of calling functions. Every point links to corresponding line on source code (see Fig. 7).

IX. RESULTS

The method was applied to a real time operating system kernel. It has been already tested and was worked several years in production. The amount of analyzed code was about 50 000 lines. We found about 20 new data races, acknowledged by developers. Total amount of warnings was 139. It takes about 3 minutes and 6 Gb of RAM for the analysis. Also there was a test launch of the tool on the Linux kernel 3.8, on `drivers/` directory. The amount of analyzed modules was about 3500.

Source code

```

Race_example.c
1 #line 1 "../cil-files/Race_example.c"
2 int global;
3
4 int print() {
5     if (global % 2 == 0) {
6         printf("global is even: %d", global);
7     } else {
8         printf("global is odd: %d", global);
9     }
10 }
11
12 int increase() {
13     lock();
14     global++;
15     unlock();
16 }
17
18 int main() {
19     switch(undef_int()) {
20         case 0:
21             print();
22             break;
23
24         case 1:
25             increase();
26             break;
27     }
28 }

```

Fig. 7. Example of source code

Error trace

<input checked="" type="checkbox"/> Function bodies	<input checked="" type="checkbox"/> Blocks	Others...
<pre> /*Number of usages:5*/ /*Two examples:*/ /*-----*/ /*Without locks*/ -main() { 21 -print() { 8 f(global) { /* Function call is skipped return ; } return ; } /*-----*/ /*example_lock[1]*/ -main() { 25 -increase() { 13 example_lock[1] 14 global = ...; return ; } return ; } } </pre>		

Fig. 8. Example of unsafe

The tool generated about 900 unsafe cases. Several of them were analyzed and one actual bug was found, but it had been already fixed in the newest version of Linux kernel.

X. RELATED WORK

In our method we are performing static analysis in contrast to dynamic analysis which has its own benefits. We are considering only methods for the analysis of C code, excluding for example Java analyzers, like [12]. The method of Locksmith [6] is the most similar. It is also based on Lockset algorithm, but has different approaches for the analysis of locks and shared data. Basically it performs intra-procedural analysis with propagation of constraints which gives it context-sensitivity, but it does not take into account path conditions. Our method is inter-procedural and explores each path separately as long as they result in different states. As far as it is based on CPAchecker it allows to extend the method with existing analysis, like explicit or predicate. Another distinction is a computation of shared data. The Locksmith determines sharedness for a variable across all program, while in our method the sharedness is determined for a particular program location.

XI. FUTURE WORK

The main problem of all methods of static analysis is a great amount of false alarms. Some of them can be discarded with help of shared analysis. But at the moment the large part of false alarms are caused by inaccuracies in analysis of expressions. For instance, now the analysis does not properly consider conditions in if statements. There is an existing method, called CEGAR (Counterexample Guided Abstraction Refinement) [14], which takes into account conditions by means of predicated abstraction, but it is used for checking reachability properties. In case of data races CEGAR algorithm should be modified to take into account that two threads should be considered instead of one. In particular, it means that it should be able to refine many error paths together.

Another issue is a full launch on Linux kernel with analysis of results.

REFERENCES

- [1] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, *A Study of Linux File System Evolution*, 11th USENIX Conference on File and Storage Technologies (FAST '13)
- [2] Mutilin V.S., Novikov E.M., Khoroshilov A.V. *Analysis of typical faults in Linux operating system drivers*. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349–374, 2012 (in Russian).
- [3] Stefan Savage, Michael Burrows, Greg Nelson, Patric Sobalvarro, Thomas Anderson *Eraser: A Dynamic Data Race Detector for Multi-threaded Programs* ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.
- [4] Gerlits E.A., Kuliain V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. *Testing of Operating Systems*. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, pp. 73–107, 2014 (in Russian).
- [5] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk *Effective Data-Race Detection for the Kernel* Operating System Design and Implementation (OSDI'10), 2010, USENIX.
- [6] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. *Locksmith: Practical Static Race Detection for C*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 33(1):Article 3, January 2011.
- [7] Polyvios Pratikakis, Jeffrey S. Foster, Michael Hicks. *Locksmith: Context-Sensitive Correlation Analysis for Race Detection*, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pp. 320 - 331, ACM New York, 2006
- [8] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, Georg Weissenbacher, *Model Checking Concurrent Linux Device Drivers*, ASE'07, November 4–9, 2007

- [9] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz, *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*, ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, Pages 391–411.
- [10] Daniel Wonisch, *Block Abstract Memorization for CPAchecker*, TACAS 2012, LNCS 7214, pp. 531-533.
- [11] Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. *Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]*. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
- [12] Mayur Naik, Alex Aiken, John Whaley *Effective static race detection for Java*. PLDI '06 Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, 41, 6, pp. 308 - 319, 2006
- [13] Jan Wen Vounq, Ranjit Jhala, Sorin Lerner, *RELAY: Static Race Detection on Millions of Lines of Code*. ESEC/FSE'07, 2007
- [14] Khoroshilov A.V., Mandrykin M.U., Mutilin V.S. *Introduction to CE-GAR — Counter-Example Guided Abstraction Refinement* pp. 219-292. (in Russian)

Minimizing the number of static verifier traces to reduce time for finding bugs in Linux kernel modules

Vitaly Mordan

Institute for System Programming
Russian Academy of Sciences
Moscow, Russia
Email: mordan@ispras.ru

Evgeny Novikov

Institute for System Programming
Russian Academy of Sciences
Moscow, Russia
Email: novikov@ispras.ru

Abstract— Static verifiers usually stop after they find a first bug in a program under analysis. This slows down the process of finding and fixing of bugs of the same kind in a given Linux kernel module. In order to solve this problem we used the static verifier CPAchecker with option to continue analysis after finding of a first bug. Besides we extended LDV Tools – a toolset for verification of Linux kernel modules – for finding several bugs in a given module against a specified rule specification. But first experiments revealed a new problem – the verifier produced too many similar traces. The given paper introduces a formal definition of equivalent traces and presents different comparison algorithms and a semi-automated approach, which makes possible to find several bugs in a given Linux kernel module against a specified rule specification at once.

Keywords—Linux kernel module; correctness rule; static verifier trace; equivalence class.

I. INTRODUCTION

The Linux kernel is one of the most fast-paced software projects [1]. The number of its active developers is more than one thousand. A new release comes out in 2-3 months, it contains thousands of changes. At present the Linux kernel consists of more than 15 million lines of code. At the same time every bug in the Linux kernel is critical [2]. Researches revealed that most of bugs in the Linux kernel are located in its modules (modules contain approximately 7 times more bugs than the kernel itself) [3]. One of the approaches to find those bugs is using static verifiers. At the moment just LDV Tools [4] among all toolsets that allow to apply static verifiers for Linux kernel modules are under active development. This toolset has already helped to find 150 bugs in Linux kernel modules and corresponding patches were applied by the Linux kernel developers [5].

Each Linux kernel module can contain any number of bugs. At the same time most of static verifiers stop after they find a first bug. One can fix this bug and repeat verification until no bugs will be detected in a given module but this approach may take a lot of time and can not be automated.

Let us consider the following program (Fig. 1). There is function `getnchar` which works properly only with positive values of its parameter. A static verifier needs to check that all

calls of this function is correct. A formal task for the static verifier is to check that label `ERROR` can not be reached.

```
1: void getnchar(int n)
2: {
3:     if (n <= 0)
4:         ERROR: goto ERROR;
5:     ...
6: }
7: int main()
8: {
9:     getnchar(-1);
10:    getnchar(-2);
11:    return 0;
12: }
```

Fig. 1. Program with 2 bugs.

There are two bugs in this program: at lines 9 and 10 correspondingly. Most of static verifiers most likely will find the first one and will stop their analysis. After fixing of this bug the user may think the program is safe but it is not the case the program still has another bug.

For simple programs this problem is not so critical. But for such big projects like the Linux kernel, where checking of all modules against one rule specification requires more than 1 day and where fixing of bugs requires preparing and applying of corresponding patches, this problem greatly increases time for finding all bugs.

This paper presents an approach to reduce time for finding all bugs in Linux kernel modules. The approach based on Linux Driver Verification Tools (LDV Tools), which are an open source toolset for checking correctness of Linux kernel modules against rule specifications with help of different static verifiers [6]. The LDV Tools architecture is presented in Fig. 2.

LDV Tools provide an interface to verify a set of modules against a set of rule specifications. When verification is complete, verification results will be placed in an archive. After that, this archive can be uploaded to a database. LDV Analytics Center allows to analyze the verification results. It is integrated with Error Trace Visualizer that visualize traces from a static verifier to allow the user either to find actual bugs or to identify source of false alarms. Also LDV Analytics Center provides an

interface to Knowledge Base that keeps information on all already analyzed unsafes. Knowledge Base scripts mark new uploaded static verifier traces if they are equivalent to any of already analyzed traces by specified criteria.

There are several examples of more than one bug for a given rule and a given Linux kernel module, which were found by LDV Tools [5]. For example, bug reports L0029¹ and L0034² are devoted to a lack of mutex release in different functions in the kernel of versions 2.6.38 and 2.6.39 correspondingly. Bug report L0030³ deals with a lack of mutex release in a couple of places and so on. These examples show that sometimes it is possible to find similar bugs if a verification toolset is able to find one of them. In order to solve this problem it was decided to extend LDV Tools so that the toolset could find all bugs in a given module against a specified rule specification.

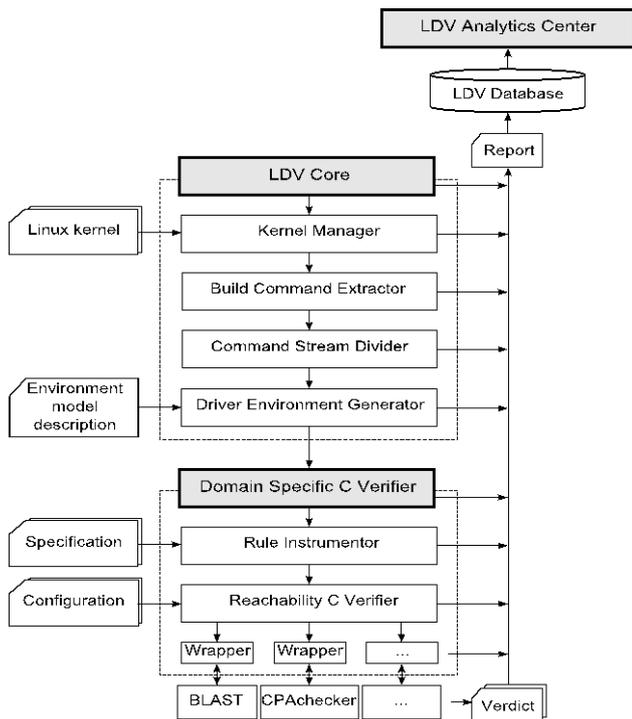


Fig. 2. LDV Tools architecture.

First, we need a static verifier that is able to continue analysis after a first bug is found. CPAchecker is a tool for configurable software verification [7]. It aims at easy integration of new verification components. With revision r8387⁴ it has an option “find and report multiple specification violations”. Also it already has been integrated with LDV Tools [6]. That is why it was decided to use CPAchecker as the static verifier for finding all bugs in a given Linux kernel module against a specified rule specification in LDV Tools.

1 <http://linuxtesting.org/results/report?num=L0029>
 2 <http://linuxtesting.org/results/report?num=L0034>
 3 <http://linuxtesting.org/results/report?num=L0030>
 4 <https://svn.sosy-lab.org/software/cpachecker/trunk/>

Second, some components of LDV Tools were improved so that they can process multiple traces (formal definitions will be given in Section II) from CPAchecker. The CPAchecker wrapper was extended to get multiple traces from the static verifier and to send them to Reachability C Verifier. Reachability C Verifier was improved to put all traces into a final report. Finally, LDV Uploader, which uploads analysis results into a database, was extended for uploading several traces for a given module.

After that we conducted experiments. At first LDV Tools with CPAchecker were checked on known issues (L0029, L0034 and L0030). 2 bugs were found for L0029/L0034 and 3 bugs were found for L0030 – as expected. More thorough evaluation of LDV Tools with CPAchecker was made on known bugs in Linux kernel 3.12-rc1 (in total 15 modules and 4 rule specifications). Then a main issue of this approach was revealed – the total number of found traces was 1998 for only 15 modules. Manual examination revealed only 23 different bugs (see Table 1), a lot of traces were similar.

In order to find several bugs in Linux kernel modules in practice the number of found traces should be minimized, but we should pay much attention to keep those traces that correspond to different bugs.

Next section gives some formal definitions. Section III describes comparison algorithms for traces. In section IV semi-automated approach was suggested to further improve comparison algorithms. In Section V the results are presented.

II. FORMAL DEFINITIONS

Below some formal definitions are given.

Static verifier trace – is a sequence of operations (assignments, function calls, assumptions, etc.) in program source files that leads from a specific entry point to a specific label. By default entry point *main* and label *ERROR* are used. Different static verifiers use different formats for their traces, so in LDV Tools all such traces are converted into the common format [8]. There are 4 types of nodes which can be used in traces in the common format:

- *CALL* – call of a specified function;
- *RETURN* – return from the last called but not yet returned function;
- *ASSUME* – choosing a specified branch in conditional clauses;
- *BLOCK* – contains assignments and some auxiliary operators (like *goto*).

There is also information on a line number, on a source file and formal arguments (for function calls) for each operation. For example, the static verifier trace in the common format for the program in Fig. 1:

```
8 "test.c" CALL      : main()
10 "test.c" CALL 'n' : getnchar(-1)
3  "test.c" ASSUME   : (n < 0)
4  "test.c" BLOCK    : goto ERROR
```

Function call tree – a tree corresponding to function calls in a static verifier trace. Hereafter we suppose that all function

calls in function call trees are ordered. It can be obtained by removing *ASSUME* and *BLOCK* nodes from static verifier trace.

LDV model functions – functions in rule specifications, which contains the main logic of correctness rules [9].

Bug – a reason which causes a violation of a specified correctness rule. For example, it is “lack of mutex release in probe function”. Fixing of a bug is usually represented as a patch.

Two or more static verifier traces are called **equivalent** if they correspond to the same bug. In other words, for all equivalent static verifier traces a reason of a correctness rule violation is the same.

Therefore all static verifier traces can be divided into equivalence classes. Each equivalence class contains only equivalent static verifier traces, i.e. corresponding to the same bug. After such dividing it will be necessary to manually analyze only one static verifier trace from each equivalence class. In this case only one static verifier trace for each bug will be in the final report and their number can be reduced without missing any bugs.

III. STATIC VERIFIER TRACE COMPARISON ALGORITHMS

In order to divide all static verifier traces into equivalence classes we propose to use special algorithms for their comparison. Ideally results of these algorithms should satisfy the definition of equivalent traces. But in practice it is often impossible to automatically understand where is a reason of a correctness rule violation in a static verifier trace.

Static verifier trace comparison algorithms should somehow reduce the number of static verifier traces, but do not remove traces that correspond to different bugs. They can take into account specifics of Linux kernel modules and correctness rules. For example, the main logic of correctness rules is in LDV model functions.

All suggested below static verifier trace comparison algorithms are not absolutely correct in terms of the given definition but in practice it is expected that the number of static verifier traces corresponding to different bugs, which were called equivalent by an algorithm, to be minimal (or even zero) because of specifics of Linux kernel modules and LDV Tools rule specifications.

A. Function call tree comparison algorithm

The first algorithm is aimed to not consider *BLOCKS* and *ASSUMES* in static verifier traces while comparing these traces. *BLOCKS* usually are used for assignments which can not lead to the *ERROR* label unless they are inside a model function. If *ASSUME* adds some new branch in a static verifier trace it will be revealed in function calls in that branch if so. So only *CALLS* and *RETURNS* from static verifier traces are considered by this algorithm.

Function call tree comparison algorithm: static verifier traces are considered equivalent if their function call trees are equal.

For example, next two static verifier traces for the same program will be equivalent:

```
CALL : function_1()
ASSUME : (x != 1)
BLOCK : x = 0;
CALL : function_2()
BLOCK : y = 1;
RETURN
```

and

```
CALL : function_1()
ASSUME : (x == 1)
CALL : function_2()
BLOCK : y = 1;
RETURN
```

This algorithm was implemented as a function in the Reachability C Verifier component. The CPAChecker wrapper gets all found static verifier traces and sends them to Reachability C Verifier as before. But before adding them into the final report Reachability C Verifier will call the function implementing the given comparison algorithm, if the specific option was specified in launching LDV Tools. This function will filter received traces and will return only the first static verifier trace from each equivalence class. After that, those traces will be added into the final report.

Unfortunately this algorithm is far from optimal since there is still a lot of equivalent static verifier traces from different equivalence classes. In experiments 1998 static verifier traces from 15 Linux kernel modules were divided into 1812 equivalence classes while the ideal result is 23.

In general case this algorithm can add static verifier traces, that correspond to different bugs, into the same equivalence class. For example, assignments and assumes outside function calls can lead to different bugs (Fig. 3).

```
1: void func(int arg)
2: {
3:   if (arg == 0)
4:     mutex_lock(&mutex);
5:   elseif (arg == 1)
6:     ...// no function calls <- 1st bug
7:   else
8:     ...// no function calls <- 2nd bug
9:   mutex_unlock(&mutex);
10: }
```

Fig. 3. Example of incorrect operation of the function call tree comparison algorithm.

There are two different static verifier traces:

```
CALL : func
ASSUME : arg == 1
CALL : mutex_unlock
```

and

```
CALL : func
ASSUME : arg != 0
ASSUME : arg != 1
CALL : mutex_unlock
```

which will belong to the same equivalence class.

B. Model functions comparison algorithm

Since the main logic of rule specifications is in LDV model functions, the previous algorithm was improved to shrink function call trees so that each their leaf is a model function call.

Model functions comparison algorithm: static verifier traces are considered equivalent if they have equal model function call trees, in which any subtree have a model function call.

Thus in order to compare two static verifier traces with this algorithm the following steps are required:

- 1) Find function call trees for both static verifier traces (like in the previous algorithm).
- 2) Determine all model functions, that were called in a module under analysis (all of them have specific comments "*LDV_COMMENT_MODEL_FUNCTION_DEFINITION*").
- 3) Delete from both trees all subtrees, that does not have any model functions calls at all.
- 4) Compare resulting model function trees.

For example, next two parts of static verifier traces will be equivalent (functions $f()$, $g()$ and $h()$ do not have any calls of model functions in their function call trees):

```
CALL : f()
RETURN
CALL : g()
CALL : h()
RETURN
CALL : ldv_func()
RETURN
RETURN

and

CALL : g()
CALL : ldv_func()
RETURN
RETURN
```

This algorithm was implemented as a function in Reachability C Verifier component similar to the function call tree comparison algorithm.

In practice this algorithm showed much better results – 1998 static verifier traces were divided into 482 equivalence classes (Table 1). For some modules (like *drivers/usb/misc/ftdi-elan.ko*) this algorithm made significant improvement – 947 static verifier traces were divided into 47 equivalence classes. But there are still cases (like *drivers/input/tablet/gtco.ko*), where this algorithm could not reduce the number of static verifier traces at all.

C. Probe comparison algorithm.

In the previous algorithms only types of nodes from static verifier traces and LDV model functions of specified rule specifications were used. But also there are some specific functions in Linux kernel modules, that also can be used in order to compare static verifier traces. Examples, where the previous algorithms failed, may provide information on what can be helpful.

More detailed analysis revealed that probe functions of modules (these functions initialize new attached devices for instances) can be called any number of times. So if this function has a bug, which will be revealed lately (e.g. memory leak), it can be called any number of times before this bug finally will be found (Fig. 4).

```
1: probe(); // probe failed
2: // ... any number of failed probe calls
3: probe(); // probe failed
4: ldv_check_final_state(); // bug will be found
```

Fig. 4. One bug in the probe function can cause a lot of different static verifier traces.

Thus we have a lot of different static verifier traces that actually correspond to the same bug. This issue should be resolved in order to reduce the number of equivalent static verifier traces.

Probe comparison algorithm: static verifier traces are considered equivalent if they are equivalent by the model functions algorithm and all sequences of calls of a probe function with the same name are considered as one function call.

Theoretically there can be static verifier traces, that are not equivalent, but were called equivalent by this algorithm. But in practice, there were no such cases so far.

The main idea of this algorithm is the following. If there was a bug in some function, there is no need to call that function again and again, a first trace already corresponds to that bug. That is why it is enough to call such the function only once. And it turned out that probe functions were often the reason of such problems.

In practice this algorithm showed the best results – 1998 static verifier traces were divided into 64 equivalence classes (Table 1). For some modules (like *drivers/input/tablet/gtco.ko*, *drivers/media/rc/imon.ko*, *drivers/staging/media/lirc/lirc_sasem.ko* etc.) the result was ideal. Only in one case there was still many static verifier traces – 35 for *drivers/usb/misc/ftdi-elan.ko*. Nevertheless it is much more easy to analyze 64 static verifier traces than 482.

It is unlikely that this algorithm will be used in practice together with the previous algorithms in the future, because it is rather specific. But for a given research it has helped to significantly reduce the number of equivalent static verifier traces and has made possible to analyze them. This revealed more common problems that can not be solved simply with comparison algorithms.

IV. SEMI-AUTOMATED APPROACH

In order to find a better solution to minimize the number of static verifier traces more thorough analysis of probe comparison algorithm results was made.

A. Probe comparison algorithm results analysis.

For 9 modules eventually results are ideal (see Table 1). 2 modules among them contains more than one bug (*drivers/media/rc/imon.ko* and *drivers/media/rc/imon.ko*). For *drivers/media/rc/imon.ko* there were 3 equivalence classes. There are 3 paths in function *imon_probe* that lead to missing of `put` after `get`. For *drivers/net/wireless/ath/carl9170/carl9170.ko* there were 7 equivalence classes which correspond to 7 different bugs (*rcu* calls inside critical section).

For *drivers/staging/media/as102/dvb-as102.ko* additionally found static verifier trace is actually a false alarm because of an incorrect environment model (release was called without probe). So this problem is beyond static verifier trace comparison algorithms.

Module *drivers/usb/misc/ftdi-elan.ko* introduces a new problem. There is one bug, that was revealed later in different places which becomes different static verifiers traces. The same problem is in module *drivers/media/usb/pvrusb2/pvrusb2.ko*. This problem is the general case of an issue with probe functions considered earlier.

Module *drivers/net/tun.ko* contains 1 bug – *hlist_add_head_rcu* call inside critical section. But it also contains 4 different traces to the given bug. The same problem (different paths to the bug) is in 2 other modules. In module *net/rxrpc/af-rxrpc.ko* an additional path with lock-unlock was found. In both cases bugs were the same. In module *drivers/net/wireless/rtl818x/rtl8187/rtl8187.ko* a second trace consists of a correct and incorrect calls of function probe. Sometimes other paths to a bug can be helpful for understanding and for fixing the bug, but we consider them excessive.

B. Revealed problems.

The distribution of reasons of diversities from ideal results on 15 Linux kernel modules for probe comparison algorithm is presented in Fig. 5.

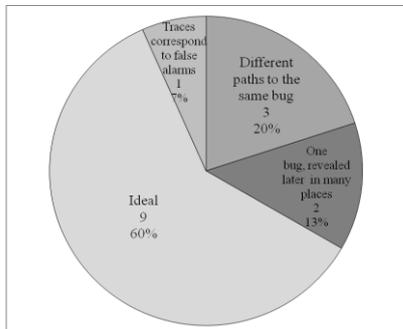


Fig. 5. The distribution of reasons of diversities from ideal results on 15 Linux kernel modules.

Static verifier trace comparison algorithms do not aim to deal with false alarms, so we are not going to consider this problem in the given paper. But other problems ('one bug revealed later in different places' and 'different paths to the same bug') should be resolved. The probe algorithm has solved the problem when one bug was revealed later in many places for one special case. But in the general case this problem can not be solved so easily.

For example, there is one bug in function probe (a lock is acquired but not released), that could be revealed in different places later (Fig. 6). As one can see there is no sequences of calls of failing function probe – they were assigned to the same equivalence class by the probe comparison algorithm. In examples below *lock* acquires locks and *unlock* releases locks.

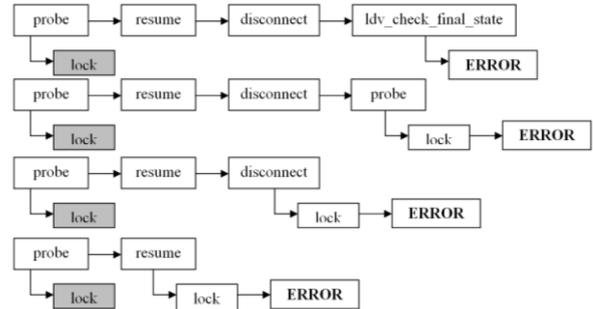


Fig. 6. 4 static verifier traces that correspond to the same bug.

At the same time there could be a situation with different bugs and different static verifier traces. For example, there are 2 bugs (release of an unacquired lock) in 2 different functions (Fig. 7).

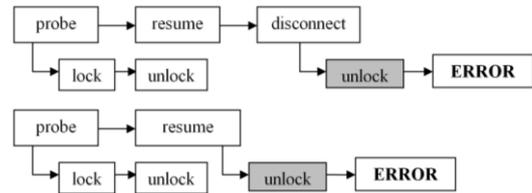


Fig. 7. 2 static verifier traces that correspond to different bugs.

C. Suggested approach.

To solve the problems presented above we suggest a semi-automated approach:

- 1) When all static verifier traces are added into the final report after some comparison algorithm was applied (for example, the probe comparison algorithm)
- 2) Then an expert marks up a some bug in LDV Analytics Center.
- 3) After that a special script finds all static verifier traces, that are equivalent to the trace corresponding to the bug, and excludes them from further analysis.

This approach was implemented as a script for LDV Tools Knowledge Base. The expert can mark up specified place in the static verifier trace (for example, function with a bug) and

use this script. The script will automatically mark all static verifier traces, that contains the same bug, among all traces relevant a given module. After that the expert can see in LDV Analytics Center which static verifier traces were considered to have the same bug and ignore them.

This approach has helped to resolve both problems. For 4 modules (*net/rxrpc/af-rxrpc.ko*, *drivers/media/usb/pvrusb2/pvrusb2.ko*, *drivers/net/wireless/rtl818x/rtl8187/rtl8187.ko* and *drivers/net/tun.ko*) only one bug was found as in ideal results (Table 1). For module *drivers/usb/misc/ftdi-elan.ko* 31 static verifier traces were marked as corresponding to the same bug, one trace is appeared to be a false alarm because of an incorrect environment model (same as above) and 3 traces are appeared to be false alarm because of pointer analysis in CPAchecker 1.2, which was used for conducting experiments (this problem was resolved in CPAchecker 1.3.4).

So, the number of static verifier traces was reduced to ideal results (if not consider false alarms) by means of the probe comparison algorithm and the semi-automated approach.

V. RESULTS

The results of all experiments are presented in Table 1. In experiments LDV Tools rule specifications 39_7a⁵, 106_1a⁶, 132_1a⁷ and 147_1a⁸ were used.

Linux kernel module	RS	FE	Tree	MF	Probe	SAA	Manual
<i>net/rxrpc/af-rxrpc.ko</i>	39_7a	6	5	2	2	1	1
<i>drivers/media/usb/pvrusb2/pvrusb2.ko</i>	106_1a	3	3	2	2	1	1
<i>drivers/input/tablet/gtco.ko</i>	132_1a	65	65	65	1	1	1
<i>drivers/isdn/gigaset/bas_gigaset.ko</i>	132_1a	17	17	2	1	1	1
<i>drivers/media/rc/imon.ko</i>	132_1a	136	76	15	3	3	3
<i>drivers/staging/gdm724x/gdmulte.ko</i>	132_1a	4	4	4	1	1	1
<i>drivers/staging/gdm72xx/gdmwm.ko</i>	132_1a	6	6	5	1	1	1
<i>drivers/staging/media/as102/dvb-as102.ko</i>	132_1a	2	2	2	2	2	1
<i>drivers/staging/media/lirc/lirc_imon.ko</i>	132_1a	394	278	15	1	1	1
<i>drivers/staging/media/lirc/lirc_sasem.ko</i>	132_1a	234	234	234	1	1	1
<i>drivers/net/wireless/rtl818x/rtl8187/rtl8187.ko</i>	132_1a	2	2	2	2	1	1
<i>drivers/usb/misc/ftdi-elan.ko</i>	132_1a	947	938	47	35	5	1
<i>drivers/usb/wusbcore/wusb-cbaf.ko</i>	132_1a	154	154	76	1	1	1
<i>drivers/net/tun.ko</i>	147_1a	17	17	4	4	1	1
<i>drivers/net/wireless/ath/carl9170/carl9170.ko</i>	147_1a	11	11	7	7	7	7
Total – 15 modules		4	1998	1812	482	64	28

Table 1. Summary table

(‘RS’ – rule specification, ‘FE’ – first experiment, ‘Tree’ – function call tree comparison algorithm, ‘MF’ – model function comparison algorithm, ‘Probe’ – probe comparison algorithm, ‘SAA’ – semi-automated approach, ‘Manual’ – manual examination).

The suggested approach has helped to find 8 new bugs in Linux kernel 3.12-rc1 modules *drivers/media/rc/imon.ko* and *drivers/media/rc/imon.ko*.

- 5 <http://forge.ispras.ru/issues/867>
- 6 <http://forge.ispras.ru/issues/2742>
- 7 <http://forge.ispras.ru/issues/3306>
- 8 <http://forge.ispras.ru/issues/3832>

VI. CONCLUSION

The suggested approach provides means to reduce the number of static verifier traces by dividing them into equivalence classes. Proposed algorithms for static verifier trace comparison show acceptable results and they allow to reach ideal results with not big efforts of the experts. This approach makes possible to analyze all bugs in a given Linux kernel module against a specified rule specification found by LDV Tools and CPAchecker static verifier.

The suggested approach should also be tested on known false positives to make sure that it also works as expected. In future this approach will be extended to check several rule specifications at once, that is very promising area of research. This will further reduce time for finding bugs in Linux kernel modules.

The suggested approach can be applied in other areas outside of Linux kernel modules and LDV Tools.

REFERENCES

- [1] Corbet J., Kroah-Hartman G., McPherson A. *Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*. <http://go.linuxfoundation.org/who-writes-linux-2012>, 2012.
- [2] Beyer D. Petrenko A. *Linux Driver Verification*. In Proc. Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies, LNCS, vol. 7610, pp. 1-6, 2012. doi: 10.1007/s10009-007-0044-z.
- [3] Chou A., Yang J., Chelf B., Hallem S., Engler D. *An Empirical Study of Operating System Errors*. In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 73-88, 2001. doi: 10.1145/502034.502042.
- [4] Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. *Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]*. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
- [5] Bugs found in Linux kernel modules with help of LDV Tools. <http://linuxtesting.org/results/ldv>.
- [6] Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. *Towards an Open Framework for C Verification Tools Benchmarking*. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17.
- [7] Beyer D., Keremoglu M.E. *CPAchecker: A Tool for Configurable Software Verification*. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16.
- [8] Novikov E.M. *Uproshhenie analiza trass oshibok instrumentov staticheskogo analiza koda. [Simplification of static verifier traces analysis]*. APSPI, 2011. (in Russian).
- [9] Novikov E.M. *Razvitiye metoda kontraktnykh spetsifikatsij dlya verifikatsii modulej yadra operatsionnoj sistemy Linux [Development of a contract specification method for the verification of Linux kernel modules]*. Dissertatsiya na soiskanie uchenoj stepeni k.f.-m.n. [PhD thesis], 2013 (in Russian).

Tools Support for Linux Kernel Deductive Verification Workflow

Denis Efremov
Institute for System Programming of
Russian Academy of Sciences
Moscow, Russia
efremov@ispras.ru

Nikita Komarov
Institute for System Programming of
Russian Academy of Sciences
Moscow, Russia
nkomarov@ispras.ru

Abstract—Errors in critically important systems may become very expensive. If such systems must provide confidentiality when working with some critically important data such as classified information or private know-how, an error cost may become difficult to evaluate. For these systems, formal verification methods should be used to prove they are error-free. In the paper, a case of formal verification of such system – a Linux kernel security module – is considered; the chosen toolset, the verification process workflow are reviewed, along with some auxiliary tools required for this process and developed by the authors.

I. INTRODUCTION

With the growth of software systems complexity, new requirements for their correctness and robustness emerge. Errors in critically important systems may become very expensive. For such systems, just a thorough comprehensive testing with maximum possible coverage is not enough; the only way to ensure the system is error-free is formal verification – mathematical proving of system correctness and compliance with requirements.

One of the formal verification bases is Hoare logic [1]. Its central part is so-called, Hoare triple, describing how an execution of code fragment changes the computation state. Hoare triple looks this way:

$$\{P\}C\{Q\}$$

where P and Q are predicates and C is a command. P is called precondition, Q is called postcondition. If the precondition is true, then running the command makes postcondition true too. So, the program is divided into such fragments, and for each of them there is some precondition and some postcondition. It can be proven that, if the precondition holds, after the execution of this code fragment, the postcondition holds too. Examining the whole program in such a way, it can be proven that, if the precondition for the whole program holds, after its execution, the postcondition for the whole program holds. Robert W. Floyd has also developed a similar to Hoare logic method at the same time, applied to flowcharts [2].

The seL4 microkernel [3] developers had a task of performing the formal, machine-checked verification of the seL4 microkernel. SeL4 is a microkernel of L4 family, and

when it was developed, one of the main requirements for it were highest security and reliability possible. SeL4 is relatively small (about 8700 lines of code in C language, plus about 600 lines of code in assembly language). Therefore formal approach to its verification could have been used, that allowed to demonstrate design correctness and prove C language implementation correctness.

SeL4 developers' approach to kernel development process is remarkable. As a first step, abstract kernel specification is developed. Based on it, high-level kernel prototype is developed, using a subset of Haskell functional language. This prototype is automatically translated into a executable specifications, and then compliance between formal and executable specifications is proven. Executable specification uses Isabelle/HOL prover language, which the developers have chosen for kernel correctness proof. As the next step the prototype was running on hardware simulator. This allows to evaluate kernel's design correctness.

Then, based on the aforementioned Haskell executable prototype, final kernel implementation is developed. C language was used. Kernel developers don't use Haskell code itself, because it would require a Haskell runtime environment for kernel to function. And this runtime environment is bigger than C kernel implementation. Also, because of using the low-level programming language, some additional optimizations and performance tweaks can be applied. After the development of the final kernel implementation, its correctness and compliance with the executable specification are proven.

There are, of course, some limitations to this approach. In particular, correctness of the parts of the kernel implemented in assembly language, which include some important parts such as MMU, is not proven. Also, correctness of compiler, boot loader and hardware is not proven either.

From some point of view, we are working on a quite similar task now. Linux kernel security module based on formal security model [4] and using standard Linux kernel security interfaces (LSM, Linux Security Modules) [5] with some extensions has been developed. This module is a part of Astra Linux Special Edition distribution, which is security-enabled and certified for working with confidential

data. The size of security module's source code is about 4500 lines. The challenge is that the module was in an active development state at the time when its verification has started. The code wasn't stabilized and was a subject to changes, including not only bugs fixing, but also adding some new features.

Because of the task's importance, our team have decided to use formal verification methods. So, the task is as follows: to formally prove the correctness of Linux kernel module implementation in C language, and to prove the compliance of this implementation with the abstract security model. The correctness of security model has been verified separately. Also the multi-threaded nature of Linux kernel environment should be taken into account. In particular, this means to proof the absence of simultaneous memory access problems such as race conditions.

The paper is organized as follows: in section II brief description of instruments used for deductive verification process is given; in section III some features of the source code of target module are described; in section IV verification process workflow is described; in section V some additional tools which are developed by the authors and useful in verification process are described.

II. TOOLSET

There are a number of deductive software verification tools. All of them are used in a similar way: functions are annotated with pre- and postconditions; each loop is annotated with invariant that has to be preserved on each iteration; one can also specify some lemmas that allow tools to prove complicated statements. For functions that aren't proven, but are used in the code under analysis, it is necessary to write pre- and postconditions only. Developing specifications for macros is usually impossible due to the fact that the tools work with the code already preprocessed. Let's consider some of such tools.

A. Verifast

Verifast [6] is a tool for verification of single- and multi-threaded programs written in C or Java. This tool has been developed in the Belgian Katholieke Universiteit Leuven. It uses a modified Hoare logic, function annotations are written in its own original language. To prove the specification conditions correctness, Z3 SMT solver is used. There is also an IDE with graphical user interface. Unfortunately, Verifast is not free software. Its source code is not open, which might cause some problems to the point of inability to use the tool in certain situations, given the task complexity and non-triviality.

B. Boogie

Boogie [7] is an intermediate language (formerly called BoogiePL) and a tool for verification of programs developed by Microsoft Research. It is language-independent, for now there is a translation support available for languages Spec#, C, Dafny, Eiffel and Java bytecode with

BML. The tool supports verification of multithreaded programs [8]. Boogie also isn't free, which might cause problems similar to those described in paragraph II-A.

C. Frama-C + Why + Jessie

Jessie [16] is designed for deductive program verification. It is a plugin for Frama-C, a platform for static analysis of programs written in C language. Frama-C is developed jointly by the two French organizations: CEA-LIST (Software Reliability Laboratory) and INRIA-Saclay (ProVal team, common with LRI-CNRS and Université Paris-Sud 11). Jessie is written in OCaml language. Most importantly, it is a free software, which allows one to fix quickly its shortcomings revealed during the tool usage. Specifications for Jessie are written in ACSL language [10]. ACSL gives the ability to develop specifications of different levels, from more abstract to more specific. Jessie uses the Why platform [9] for the purpose of verification conditions translation to the format required by the specific SMT solver. It supports a wide variety of output formats, including Coq, PVS, Isabelle/HOL, Simplify, Alt-Ergo, CVC3, CVC4, Z3 etc. To solve the problem, this particular toolset has been chosen.

Unfortunately, just the formal verification tool isn't enough to solve the kernel module verification problem. First of all, the tools lack some features needed to verificate the module; for example, Jessie didn't support function pointers and variable-arguments functions at the time of project start. Secondly, some other auxiliary tools are needed to simplify the task and to establish a more efficient collaboration between the specifications developers. For example, it was found out that time required for Jessie to start may become unacceptably long when working with large source code chunks such as the security module with all the Linux kernel headers it uses.

III. TARGET MODULE SOURCE CODE ANALYSIS

Initially, the security module source code has been developed without any clear plans of its subsequent verification, or even the conception of how this can be done. Therefore, the code hasn't been limited to any C language subset. The module has been developed based on the optimal performance, not easiness of verifiability. Thus the code uses some GCC extensions of C standard actively. Moreover, because the security module is based on the Linux kernel code base, some of the code design pattern and language extensions are imposed and not always possible to give up.

So, first of all, the security module source code has been reviewed in order to identify all the features that are impossible or very difficult to work with using the verification tools. This often includes all the extensions of the C language standard, and also a well-defined set of language features [11]. This language features set may be implied by the specific verification tools, but often they are just restrictions on the possibility to prove the code

compliance with the specifications. Some coding standards include all of these limitations in advance [13] [12].

All the identified features were analyzed in respect to the possibility to avoid any usage of them in each case. Unfortunately, it was not always possible because the security module code is based on the Linux kernel code. And if some of these features are found in the kernel header files, the denial can be impossible at all. (It should be noted that verification tools often just cannot parse some unsupported language features and stop working when encounter them.) In such cases, decisions were made to add limited support of these features to verification tools. In particular, such decisions have been made on function pointers, variable-argument functions, `asm goto`, `_Bool` type support etc.

The results of this analysis of the security module source code were reported to its developers, with the emphasis on the parts that cannot be verified and need rewriting. And the parts that need to be simplified, since their proving is an almost impossible task, as well as some recommendations on coding style.

Since the verification is performed not for the entire module, but only for the part of it that is described by the mathematical model [4], some patches have been added to the project source code, introducing the preprocessor directives to make the module conditionally compile. Initially the module source code has been split into several subsystems. But no clear distinction between the part that is based on formal model and the other parts that provide additional functionality (such as logging or system calls audit subsystems) was made. These patches make the module source code easier to work with both for the verification tools (they work with the preprocessed source code) and the specifications developer. It should be also noted that the deductive verification tools are designed to work with code size of about tens of thousands of lines, and any reduction of this code base is significant for them. The security module consists of about 10 thousand lines of code, but only half of them correspond to the part that is based on formal model and requires verification.

As a next step, the authors analyzed the security module source code from the verification works plan development point of view. As the specifications are just pre- and post-conditions for functions, the module functions were analyzed with respect to the frequency of their use, their call dependency, kernel functions used by them. Although the kernel functions are not verified, preconditions and postconditions still have to be developed for them.

Macros were also investigated. Here it is necessary to provide some additional clarification. Verification tools work with the preprocessed code only. Therefore, writing specifications for macros is impossible. However, macros can be used in the bodies of specifications themselves, if they do not violate their syntax. For example, such macros are those replaced by the constants or references to the structures fields. All the macros in project were classified

into those that can be used in the specifications bodies and those that can't. For the second type, a recommendation was given to the developers to rewrite them as inline functions.

To collect the aforementioned information on the functions and macros of kernel and security module special software was developed. Its detailed description is given in the section V.

Function code can not be proven to comply to its specifications until all the functions it calls are verified too. Accordingly, the process of code verification starts from the bottom up to the top of the call graph. We made a map (Figure 1) of the module source code based on the module functions call graph. A special program was developed for this, a detailed description of which is given in the section V.

The map shows the amount of work to be done, provides an opportunity to develop the well-founded verification plan and helps to coordinate the people involved in verification process. In addition, it has immediately allowed to identify several errors in the conditional compilation directives and to find some sections of code that are most difficult to verify.

Based on the data that we got after the analysis of the security module source code, all the module functions were assigned one of 5 priority levels for specification. These levels were developed based on the current and planned language features support by instruments, so that the functions that use some unsupported features would be verified later, with respect to functionality and safety significance of individual sections of code, effort and functions dependencies.

IV. VERIFICATION PROCESS WORKFLOW

The process of kernel module source code deductive verification is as follows. As a first step, the specification for the function is developed. Secondly, the developer attempts to prove correctness of this function. Then some errors may be found in the specification and/or the source code. After that, the specification and the source code are reworked to the point when the specification can be proven. It should be also noted that the specification development, its editing and the proof are carried out in a different programs.

Currently, the verification tools can not cope with the full source code of the module (about 4500 lines of code) along with all the required kernel header files (about 70000 lines of code), because of both their size and the presence of number of language features that are unsupported for now. We can state with certainty that in the future the unsupported language features problem will be solved. However, launching the tools on the full source code will still take considerable time, and it greatly complicates the specifications proof process, because their constant refinement with external tools and, accordingly, frequent restarts of proof tools.

To reduce these difficulties, the authors have developed the following workflow of the verification process:

- 1) Fetching all the dependencies of the module/kernel code for the function to be proven. This means collecting all the data necessary to create a separate object file, with the inclusion of definitions of all functions necessary, not just their declarations.
- 2) Developing and proving the specifications. This work is carried out with the code obtained on the step 1.
- 3) Transferring the specifications back to the security module source code after they have been proven.
- 4) A full re-proof, carried out on the whole security module source code. This, for example, is needed if the specification for one of the functions fetched on the first step has changed, so that the other functions in the security module can still be proven. Also, this is needed to make sure that the preconditions for the proven functions are held at the points of their calls.

Previously, all the work was conducted manually. Later, some instruments have been developed to perform the first and the third steps of the workflow. The last step of the workflow hasn't been executed yet, because the verification tools are now being adapted to run on the full security module source code.

V. TOOLS SUPPORT

A. A tool for source code map building

There is a number of tools that allow a developer to simplify source code navigation. Some of these tools, such as doxygen [15] and cscope [14], are also able to build call graphs. However, in the case of our project, we are faced with the fact that these tools don't work correctly with the source code of the kernel module, because it's a part of a bigger project. It is impossible to build a call graph containing only module functions, but not containing kernel functions, with them. The whole graph including kernel functions would further complicate the picture, because the number of nodes and edges in this graph would prevent it to be displayed with clarity. Furthermore, these tools are not able to display the entire call graph at once, only in parts.

Because of the aforementioned tools weaknesses, it was decided to develop a software for the construction of function call graphs for Linux kernel modules. You can see the graphical result of its work on Figure 1.

From the perspective of program algorithm, it may be noted that the program works with preprocessed source code. The first step is building an index of all the functions in the module source code. The second is the analysis of indexed functions names occurrence in functions bodies and the constructing of the call graph. The third is setting some additional attributes to graph vertices (such as colors marking their belonging to the priority queues). The fourth stage is the output of the built graph in the DOT format. To build a graphical representation of the graph,

dot program from the *graphviz* package [17] is used. The developed program imposes a restriction of the function name uniqueness in the source code.

B. A tool to gather the statistics of Linux kernel functions and macros usage

The second tool, that we use to simplify the coordination of verification efforts, project planning and management, is a mean for tracking the kernel functions and macros used by security module. These dependencies cannot be displayed on the map, as there are a number of them and their display only decrease the map's clearness.

Because the task is quite unusual, our team failed to find the software which provides a turnkey solution. Thus another software has been developed.

The software works with an unpreprocessed source code of the kernel module. It scans the entire source code, that falls under a template of function call. Then C language keywords, the security module functions and macros are removed from the collected data. The list of functions that kernel exports is created by the kernel's build system. The functions that are not in the kernel's export list, but remain in the data after filtering, are considered as static inline functions and are taken into account along with the others. Kernel's macros list is created using the C preprocessor's ability to dump all the macros definitions that it encounters during its operation. The software's final output is a table of kernel functions and macros ordered by the frequency of their calls. Also, the functions that already have a specification developed are labeled in this table.

The first two tools considered, one for working with the module source code structure and another for its external dependencies analysis, have been created as aids for the verification process. The need to create them arises from the fact that security module's source code isn't stabilized yet and is under development. This development also includes regular adaptation to the new Linux kernel releases. The tools mentioned above allow the developers to track and observe these changes, and to adapt the already developed specifications in accordance with them.

C. Slicer

During the specifications development, it's often needed to review a large amount of code at once. Constant switching between different files creates a distraction for a specifications developer. However, a function, its code and data dependencies are not always localized and located close to each other. Additionally, verification tools performance depend dramatically on the amount of code they are run on, independently of whether there are specifications for this code and even whether it's really in use.

These considerations has encouraged the authors to use a tool that would extract all the code and data dependencies for the given function from both the security module source code and the kernel header files. A study

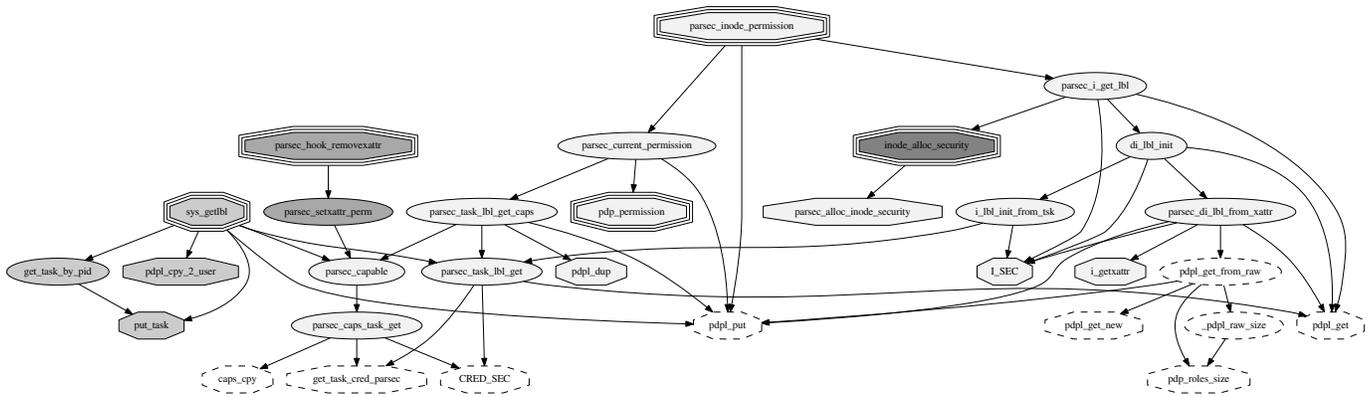


Figure 1. Map sample

of existing open source C code slicing tools such as [18] and [19] has revealed that these instruments are not useful when working with the Linux kernel source code because of some non-standard extensions present in it. However, it isn't required for the task to analyze the possible paths of the program execution. The main requirement is to keep the original functions code structure, because its change would significantly complicate the reverse specifications transfer to the module source code.

The authors have developed a software that works with an unprocessed code without performing its full parsing. It extracts macros, structures, functions, typedefs etc. definitions and declarations based on some heuristics. It builds a global graph of these objects then by searching each object's identifier in the code of the other objects. After the graph cycle resolution procedure, the code corresponding to the graph vertices that are predecessors of the particular function is output in sorted order. The result is just one C source file with functions and several C header files: one for kernel data, one for module data, and one for the kernel specifications library. Information contained in these files is enough for the compiler to build an object file.

Despite the fact that the software is based on a number of heuristics and the output files may include redundant data in some cases (for example, if a function and a structure have the same name, they would be output, regardless of whether they are both used in the code), the software lets one to get an adequate result within a reasonable time.

D. A tool for the specifications transport

Code specifications are written as simple C comments before function declarations or definitions. After the function specification has been developed, it is required to transport it back to the full kernel module source code. This allows other developers to use this specification for their specifications development process.

We have created a software that transports the specifications from one code version to another. The software

processes the two source code collections with function granularity, so its work doesn't depend on which file is which part of code located in. In the case of some specifications are already present in the old code, they are replaced with the new ones. The software recognizes the differences between code versions caused by the conditional compilation directives in the first version of the code, and automatically takes into account the absence of this code part in the second version. In the case when there are some other changes in the second part of the code, they are automatically transferred to the first. Additionally, the patch is created in this case, which can then be sent to the security module developers. When there is a conflict caused by the too many differences between code versions, which the program cannot resolve by itself, it starts an external code merge software (meld, kdiff3) to resolve them by hand.

This program is used by the authors not only as part of the verification process workflow, but also when getting a new release of the security module source code.

VI. CONCLUSION

The paper considers the organization of Linux module deductive verification process. Verification is performed in the conditions of continuing developing process of the module's code and in the absence of requirements to code written in a formal way.

During the verification activities the authors had to face restrictions of deductive verification tools and an inability to completely follow certain standards of safe coding.

The success of the code verification depends on clear organization and coordination of work. The authors have developed an approach that allows to mitigate a shortcomings of deductive verification tools and facilitate the development of specifications from the standpoint of ease of reading and analyzing the structure of the code by the developer.

Tool support necessity of worked out verification process workflow and the absence of turnkey solutions led to the development of additional tools. These tools have been

used successfully by the authors and bring the results in the form of verification problem solving approach systematization and workflow stabilization.

REFERENCES

- [1] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, Communications of the ACM, Vol. 12, Num. 10, October 1969.
- [2] Robert W. Floyd, *Assigning Meanings to Programs* Proceedings of Symposia in Applied Mathematics, Vol. 19, 1967.
- [3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, *seL4: Formal Verification of an OS Kernel*, Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2009.
- [4] Pyotr Devyanin, *The Administration of System in Course of Mandate Entity-Role DP Access and Data Flow Control Model within Linux Family OS*, Mathematical Basics of Computer Security, Issue 4 (22) 2013, in Russian.
- [5] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, Greg Kroah-Hartman, *Linux Security Modules: General Security Support for the Linux Kernel*, USENIX Security 2002.
- [6] Bart Jacobs, Frank Piessens, *The VeriFast Program Verifier*, Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, August 2008.
- [7] K. Rustan, M. Leino, *This is Boogie 2*, Microsoft Research, Redmond, WA, USA, June 2008.
- [8] Shuvendu K. Lahiri, Shaz Qadeer, Zvonimir Rakamari, *Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers*, Computer Aided Verification '09, February 2009.
- [9] Francois Bobot, Jean-Christophe Filliatre, Claude Marche, Andrei Paskevich, *Why3: Shepherd Your Herd of Provers*, Boogie 2011: First International Workshop on Intermediate Verification Languages, August 2011.
- [10] *ACSL: ANSI/ISO C Specification Language* CEA LIST and INRIA, 2009-2013.
- [11] Gerard J. Holzmann, *The Power of Ten – Rules for Developing Safety Critical Code*, IEEE Computer, June 2006, pp. 93-95.
- [12] *PL Institutional Coding Standard for the C Programming Language*, Laboratory for Reliable Software (LaRS) at the Jet Propulsion Laboratory, California Institute of Technology, March 2009.
- [13] *Guidelines for the use of the C language in critical systems*, Motor Industry Software Reliability Association (MISRA), MISRA-C: 2012, March 2013.
- [14] *Cscope – a developer’s tool for browsing source code*, <http://cscope.sourceforge.net/>
- [15] *Doxygen – generate documentation from source code*, <http://www.stack.nl/~dimitri/doxygen/>
- [16] *Krakatoa and Jessie: verification tools for Java and C programs* <http://krakatoa.lri.fr/>
- [17] Emden R. Gansner and Stephen C. North, *An open graph visualization system and its applications to software engineering*, Software – Practice and Experience, 2000, vol. 30, num. 11, pp. 1203–1233
- [18] *Frama-C - Slicing plug-in*, <http://frama-c.com/slicing.html>
- [19] Wisconsin Program, *Slicing Tool 1.1 Reference Manual*, http://www.cs.wisc.edu/wpis/slicing_tool/slicing-manual.ps

Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog

René Haberland Sergey Ivanovskiy

Department of Software Engineering and Computer Applications
Saint Petersburg Electrotechnical University "LETI"
Saint Petersburg, Russia

Abstract—A Prolog-based framework for fully automated verification currently under development for heap-based object-oriented data is introduced. Dynamically allocated issues are discussed, recent approaches and criteria are analysed. The architecture and its components are introduced by example. Finally, propositions to further and related work are given.

I. INTRODUCTION

The main interest of this work is dedicated to the correctness of a program according to its memory consumption behaviour. It may, however, also be extended to performance considerations based on results coming from the dynamic memory verification, particularly but not only during the alias analysis [17], [3] or during the garbage collection phase, for instance. A Prolog-based verifier is suggested.

The structure of this paper is to give a short overview on current approaches in section 2. Section 3 presents the languages being used as programming and specification languages, and introduces the overall architecture. The architecture is designed to be open. Currently the project is under progress, the final section gives an outlook on related and future work.

[9] discusses why the aliasing issue has still not been solved yet. [16] provides a more detailed mark on the issues related to aliasing issues in a commercial Unix-environment. Despite its age [16] and its partial closeness because of the commercial background, it is still often cited in very recent publications and numerous technical reports on the same topic, and in open-source projects, and surprisingly enough most of the issues found initially are still there almost with no changes. One key aspect that makes aliasing such a hard issue is that its local changes in a program listing may effect other regions unexpectedly – but at the same time this is its strength, since no additional copying is required. [16] had particularly analysed previously fixed bugs for a long-term period over last releases and found out - according to the bug distribution over time - that undesired memory behaviour is one of the most expensive bug reasons in terms of time and efforts to locate and fix.

Object-orientation [1], based on concepts such as encapsulation, polymorphism and inheritance, has been one of the most successful and widely adapted programming paradigms by now for a long time in industry, hence its

combination with pointer structures remains a relevant research task up to date [14].

Prolog [27] is considered for program verification for several reasons. First, it is a logical and declarative programming language which offers a high abstraction in writing Horn-clauses as they correspond with defined axioms and rules. A proof tree occasionally insists on a back-tracking strategy, which Prolog supports for free as one of its core-language features. The hope is Prolog's *generate-and-test* goal strategy [27] may be found useful in simplifying and abstracting a proof significantly. Second, programs and internal states can be represented as terms. Terms can be easily processed in Prolog. The hope is, abduction and general symbolic term evaluation will allow generating lemmas more efficiently and make the reasoning terminate and terminate earlier. As a previous successful verification attempt, [12] shall be noted. The authors solved a fairly hard problem from mathematical numerics using Prolog elegantly and straight. Since proofs are rule-centric, a proof contradiction will eventually help generating counter-examples easily by simply matching terms from the memory and from a rule or axiom.

Example1 – memory leak

```
MyClass object1=new MyClass();  
...  
object1=new MyClass();
```

Example2 — unachievable memory

```
// object1 has been created  
MyClass object2=new MyClass();  
object2.ref=object1;
```

Example3 – invalid memory access

```
// object1.ref==null  
value = (object1.ref).attribute1;
```

Example4 – data structure with cycle

```
object1.next=object1;  
...  
root=object1;  
while(root.next!=null){  
    printf("%d", object.data);
```

```
    root=root.next;
}
```

Example 1 demonstrates the case where a fresh memory region is allocated, and without freeing it, allocates it again, which may cause the previously created region becoming unachievable. The second example demonstrates where `object2` is linked to an occupied `object1`, but `object2` itself remains unused. A very common problem in practice might be the third example, when an object reference is not set, but later referenced causing either an abnormal runtime failure or continues execution, which might be even worse in realistic scenarios because the further program execution becomes totally unpredictable with invalid value settings. The fourth example might not immediately be seen as a problem, but if, for whatever reason, there is a cycle in `root` the program will not terminate. Apart from direct consequences like crashes or non-termination, one more side effect is there are spontaneous allocations/deallocations taking place on runtime which may eventually become a performance bottleneck.

II. CURRENT APPROACHES

One approach to verify correctness of dynamic memory is to get the program run and to record all memory cells that will be referenced and allocated/deallocated. This is what the *valgrind* tool [29] does. This open-source tool requires on compilation guarding memory-checking code is injected to the assembly code. Not only that the enhanced program runs with huge delays, the general problem underneath this approach as well as SAFECODE [25], which on runtime checks whether programmer-inserted assertions are fulfilled, is that only a small subset of all possible execution paths can be tested and that it requires additional code is inserted. For this reason, only static approaches are considered further that analyse the incoming program listing prior to running it.

In order to address the problems mentioned in the introduction part, several approaches exist: (i) Shape-based Analysis [26], [22], (ii) Separating heap. For sake of completeness a heap-free alternative proposed by (iii) Tufte and Talpin [28] and Meyer [15] shall be mentioned, who both appeal to a stack-based approach, if any possible, to avoid expensive heap allocation and deallocation operations. Automatic handling of stack-based locations is essential for both, where the stack sizes are determined during compilation. Thus, control passing which happens during a call will allow to allocate objects almost for *free*, since a stack frame needs to be created in any case, and no more expensive heap operations are needed in fact, for instance garbage collection. The disadvantage of (iii) is, however, there is often a platform-dependent restriction on stack sizes and number of entries, so in practise there are tough frame restrictions, e.g. a maximum offset, which should not be exceeded without getting a severe performance penalty on concrete target architectures at the same time. Meyer [15] asks to turn garbage collection steadily on during program execution. This implies for efficient execution runtime critical

parts will not trigger dynamic memory operations and those operations are opted out as efficiently as it can possibly be done.

Approaches (i) and (ii) are similar, both describe the memory state, although (i) describes the entire dynamic heap as an entire graph, where edges are region dependencies and vertices are locations. The problem with (i) is *locality*, because if a particular function is called, the entire graph has to be specified before, after and during the call, where approach (ii) allows to hide all non-affected heaps (*framed heaps*) this is what is meant by *locality principle* in terms of *Separation Logic* [24]. The most important concept behind Separation Logic [24], [23], [6] is the specification of two non-interleaving memory regions. Heaps might be composed, and programs may change heaps. If two heaps are connected, then the dependency has to be added explicitly to a current heap's specification. If a heap depends on some other heap data, then this is called *aliasing* (or *big brother property* as found in [15]).

The first implementation which makes use of Separation Logic is Smallfoot [5], [6]. In order to extend deductive reasoning capabilities, an abductive approach was proposed, called bi-abduction [7], for Separation Logic, which is a constructive guess of unchanged heaps by a greedy symbolic table-construction algorithm that chooses bigger rules first. The extension of Smallfoot is called *SpaceInvader*.

Hurlin extends in [10] the classic Separation Logic proposed in [23] by classes for a Java-like language. He suggests a *heap factorization*, an attempt to normalise heaps in order to remove redundant heap specification fragments which are considered as *noise*, even if the main goal of his thesis is focused on multi-threaded applications. He re-uses the same concept of *abstract predicates* as it was introduced by [20], [8], and generates unchanged parts during deduction with a parallel algorithm.

Parkinson [20] introduces a Java-like language with object-orientated features. Nevertheless, many problems are not being addressed yet: abstraction mismatch on encapsulation and inheritance, particularly, the problem of expanding specifications in subclasses seems to be a real hinder in simple and elegant specifications. The most essential contribution of [20] is the introduction of *abstract predicates*, although there are currently tough restrictions concerning expressibility. Super calls, static fields, reflection, inner classes and quantified predicates, for example, are currently missing language features.

Verifast [11] is another forward verifier based on Separation Logic. In comparison to all previously introduced verifiers which do very similar operations on the heap, all introduced conventions per tool differ strongly, and it does not automatically process loop invariants nor predicates - here it depends entirely on user-interaction or requires explicit injections within specification annotations inside the program which are used as internal reorganisation commands.

In [2] objects as class-instances are treated as records, typing and verification rules are introduced and a soundness proof is provided. Problems which neither in [2] nor [1] are addressed are that objects may have references to other objects

and that a lack in abstraction causes a dramatic increase in specification length which makes it in practice impossible to read and understand specification to a full extent. The memory state is specified by temporal predicates and a result-register for the previous computation step's result. There is no general recursive definition allowed, although [13] attempts to relax this hard restriction by an algebraic ideal-construction. Still there are hard restrictions, such as no aliasing nor late binding at all, and object-records only which even may become unsound for eager type evaluation.

III. ARCHITECTURE AND DESIGN

Before going on with more details on the architecture, the prerequisites on the architecture shall be summarised. The proposed architecture may be considered for teaching purposes in the future:

- 1) Automatic proof. The program and its annotations shall be sufficient in order to get the verification run. If there is an endless cycle in the proof however, there shall be no mandatory recognition, since termination is beyond the main focus of this work.
- 2) Openness. The provided architecture shall be open for extensions and variability, and the attached models shall be exportable, so it might eventually be passed through to another arbitrary model transformer, if needed.
- 3) Extensibility. The target language shall be fixed but interchangeable with an imperative programming language in the front-end. The rules and user-defined predicates shall be designed amendable, so the user may want to add rules directly to the rule set.
- 4) Plausibility: There shall be configurable visualisation facilities, so the incoming annotated program may be retrospected on each stage of the verification process. If, for instance, a proof fails or stops abruptly the user would perhaps like to see the proof tree and a counter-example.

In figure 1 the architecture of the Prolog-based verification system is shown. The input is a C-program with object-orientated extension that is annotated with assertions specifying the dynamic memory. The shortened syntax can be described by the Extended Backus-Naur form in figure 2. Not mentioned definitions as actual parameters, blocks, class methods, variable declarations have been skipped here for the sake of readability and follow mostly ANSI C. For readability purposes the expression sub-grammar has not been expanded according to its precedence hierarchy nor for optional assertions. `new` and `delete` reserve/free new chunks in the heap associated with previously defined locations. The access to heap memory is performed by [`<location>`], where `<location>` denotes either a field variable, another object's field or a either of those with an offset in order specify non-aligned memory regions, for instance. The rule `<funcall>` denotes the syntax for a method call, which may have a object specifier optionally and a method name which is required to exist with the matching total number and types of parameters being passed as expressions.

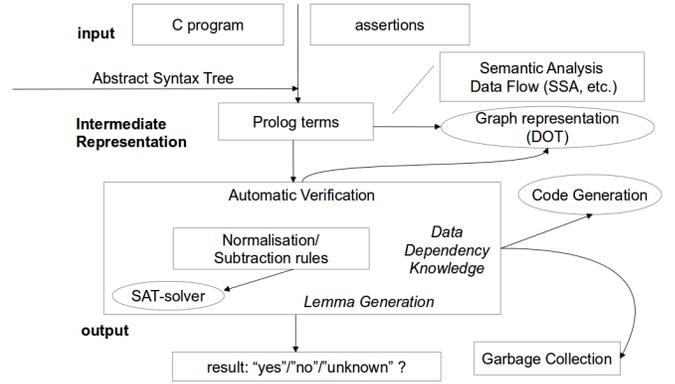


Fig. 1. Verification architecture for Prolog-based reasoning on dynamic memory

C-programs are annotated by assertions which are injected as usual Prolog terms into blocks. Blocks are encoded as lists of statement-terms. Assertions are inductively defined and can be found in figure 3. Keep in mind the expression might request object references and $\alpha(\vec{p})$ assumes predicate named α was defined prior to using it, and \vec{p} contains as many actual parameters as the arity of predicate α require there are.

```

<prog> ::= <class> <id> '{' { <field> | <method> } '}'
<location_I> ::= <id> | <id> '.' <id> | 'this' '.' <id>
<location> ::= <location_I> [ ( '+' | '-' ) <int> ]
<stmt> ::= <lhs> '=' { <lhs> '=' } <expr>
| 'if' <cond> <block> [ 'else' <block> ]
| 'while' <cond> <block>
| 'new' '(' <location_I> ')'
| 'delete' '(' <location_I> ')'
| <func_call>
<lhs> ::= <location_I> | '[' <location> ']'
<cond> ::= <expr> <rel> <expr>
<rel> ::= '&&' | '||' | '==' | '!=' | '<=' | '>=' | '>' | '<'
<expr> ::= <expr> ( '+' | '-' | '*' ) <expr>
| '.' <expr>
| '[' <location> ']'
| [ ( 'this' | <id> ) '.' ] <id> '(' <act_params> ')'
| <location>
| <int>
<func_call> ::= [ ( 'this' | <id> ) '.' ] <id> '(' <act_params> ')'

```

Fig. 2. Syntax definition of C-programs with object-oriented extension

For example,

```
int f(int a, int b) @ a<10 @ {
  id=2; a=1; b=6;
} @ a->5 * b->c * c->object(myClass1,15) @
```

is transformed into this Prolog-term:

```
function(f, int,
  [param(a,int), param(b,int)],
  [assert(le(a,10)),
  assign(id,2), assign(a,1), assign(b,6),
  assert(a->5 * b->c *
  c->object(myClass1,15))])
```

An important specification fragment of the intermediate Prolog-term syntax can be found in figure 4, where the remaining part is close to the syntax of figure 2. Apart from 'ite' which represents a *if-then-else*-construct with at least one block for the if-case and one more optional block for the else-block – as long as it was provided, there are also while-loops and further constructs, like class definitions, which will not be mentioned here for simplicity purposes. All expressions, particularly with binary operators, are encoded as terms where the literal operator becomes the functor, for example `add(i, 7)`.

$H ::=$	emp true false	<i>atomic formulae</i>
	$x \mapsto E$	<i>location map</i>
	$H * H$	<i>heap separation</i>
	$H \vee H$ $H \wedge H$	<i>conjunction</i>
	$\exists x.H$	<i>quantification</i>
	$\alpha(\vec{p})$	<i>predicate unfold</i>
		where
	x is a location	
	E is a well-defined expression (enumeration)	
	\vec{p} is a comma-separated parameter vector	

Fig. 3. Syntax definition of heap and stack assertions

Since the architecture is designed flexible, it allows the user to interchange the compiler front-end for a different language, so the user has the possibility to write own Prolog-terms directly without even having an ordinary C-program. In

```
<stm> ::= ... | 'new' '(' <loc_I> ')'
| 'delete' '(' <loc_I> ')'
| 'funcall' '(' <id> [ ',' <act_params> ] ) | ...
| 'ite' '(' <cond> ',' <block> [ ',' <block> ] )'

<loc_I> ::= <id> | 'oa' '(' <id> ',' <id> )'

<loc> ::= 'offset' '(' <loc_I> [ ',' <offset> ] )'

<offset> ::= <int> | 'minus' '(' '0' ',' <int> )'

<expr> ::= ( 'add' | 'sub' | 'mul' ) '(' <expr> ',' <expr> )'
| 'mem' '(' <loc> )'
| <loc> | <int>
| 'funcall' '(' <id> [ ',' <act_params> ] )'
```

Fig. 4. Syntax definition of Prolog-terms

this case syntax and semantic constraints remain on full responsibility to the user. Prolog-terms are internally checked and may also be directed to a graphical output, e.g. for proof tree visualisation. Antlr 4 [21] is currently used as compilation front-end.

Once the Prolog term is constructed, it can be passed to the verification. Hereby, the term is now processed while the internal environment, which has to keep the states of the memory, needs to be updated after every statement. All locals are residing in stack, where dynamically allocated memory locations may remain in memory – even if a stack-based variable stores a dynamic address it would be freed at the end of a block.

If we decide to specify a list concatenation of two lists, we have several opportunities to describe the heap. If `list(s, e)` denotes a heap predicate where s is the location of the beginning root element of a list, and e denotes the last element in that list, then having two lists x and y with $x \rightarrow a, b, c$ and $y \rightarrow d, e, f$ will concatenate for instance to either (i) $x \rightarrow a, b, c, d, e, f * y \rightarrow f$ or to (ii) $x \rightarrow a, b, c * y \rightarrow d, e, f * z \rightarrow a, b, c, d, e, f$. Remark: The ','-operator is defined as a list constructor with variable input amount for all consecutive objects currently in memory linked together to a simply-linked list [23]. The main difference between (i) and (ii) is that (i) requires only a single assignment if the end of x is known, therefore x and y are no more as they used to be before concatenation. (ii) creates an entirely new copy of all element from x and y and does not touch neither x nor y . (ii) is safer from a general reuse perspective, but it is considerably slower and consumes more memory due to additional copies to be generated.

Finally, the SMT-solver is required whenever taking out trivial calculations, for instance in basic arithmetics. For instance, if there is an expression that might be reduced to a value, then this should in general be tried first before triggering a certain rule. Beside finding solutions to basic arithmetic and other theories, re-arrangement needs to be taken into consideration. Formal rules will usually also not deal too much about heap permutation, although a strategy must be found and is crucial in fact for the overall performance.

IV. CONCLUSION

So far the open architecture was presented providing several suggestions for further research activities. Prolog was proposed as specification and proof platform for memory-specific research, e.g. on extending the expressibility of abstract predicates or abduction. We believe, questions related to abduction in Separation Logic with objects still have not been profoundly investigated yet, as well as some object-oriented features like polymorphism in Separation Logic.

The platform might be used to incorporate with existing compiler packages in order to research improvement on code optimization during the alias analysis phase, but also garbage collection, based on knowledge obtained during the dynamic memory verification.

Further rules of normalisation and re-arrangement will be applied to cover more real world scenarios, particularly in order to resolve arithmetic equivalency by the integration of a SMT-solver ([18], [19]).

Related work includes Jacobs [11] who suggests to investigate Banerjee's Regional Logic approach [4] as substitute for the Symbolic Execution approach [6].

Moreover, it includes Birkedal's approach of nesting heap specifications recursively, and code updating while program execution – this both might be of interest in a highly dynamic environment where unknown code needs to be embedded into a program interpreting external code provided as a string on runtime hence will not be considered any further in terms of this work.

REFERENCES

- [1] Abadi M., Cardelli L. A. *Theory of Objects*. New York: Springer, 1996, 396 p.
- [2] Abadi M., Leino K. R. M. *A Logic of Object-Oriented Programs*, Proc. of the 7th Int. Joint Conf. CAAP/FASE on Theory and Practice of Software Development, Springer, 1997, pp. 682-696.
- [3] Allen R., Kennedy K. *Optizing Compilers for Modern Architectures*. 2001, 790p.
- [4] Banerjee A., Naumann D. A. and Rosenberg S. *Regional logic for local reasoning about global invariants*. ECOOP, LNCS 5142, 2008, pp. 387-411
- [5] Berdine J., Calcagno C. and O'Hearn P. W. *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*. FMCO, 2005, pp. 115-137.
- [6] Berdine J., Calcagno C. and O'Hearn P. W. *Symbolic Execution with Separation Logic*. APLAS, 2005, pp. 52-68.
- [7] Calcagno C., Distefano D., O'Hearn P. and Yang H. *Compositional shape analysis by means of bi-abduction*. Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2009, 36, pp. 289-300 .
- [8] Distefano D., Parkinson M. *jStar: Towards practical verification for Java*. OOPSLA, 2008, pp. 213-226.
- [9] Hind M., *Pointer Analysis: Haven't We Solved This Problem Yet?* ACM, PASTE'01, 2001, pp. 54-61.
- [10] Hurlin C. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD Thesis, Universit Nice - Sophia Antipolis, 2009, 195p.
- [11] Jacobs B., Piessens F. *The VeriFast Program Verifier*. Leuven University, 2008, 5p.
- [12] Koch H., Schenkel A., Wittwer P. *Computer-assisted Proofs in Analysis and Programming in Logic: A case Study*. SIAM Review, 1996, no.4(38), pp. 565-604.
- [13] Leino K. R. M. *Recursive object types in a logic of object-oriented programs*. Nordic J. of Computing, (5), 1998, pp. 330-360.
- [14] Meyer B. *Proving Pointer Program Properties - Part 1: Context and Overview*, Journal of Object Technology, no.2(2), March-April 2003, pp. 87-108.
- [15] Meyer B. *Proving Pointer Program Properties - Part 2: The Overall Object Structure*, Journal of Object Technology, no.3(2), May-June 2003, pp. 77-100.
- [16] Miller B. P., Fredriksen L. and So B. *An Empirical Study of the Reliability of UNIX Utilities*. Proc. of the Workshop of Parallel and Distributed Debugging, Digital Equipment Corporation, 1990, pp. 1-22.
- [17] Muchnik S. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 2007, 856p.
- [18] Nanevski A., Morrisett G., Shinnar A., Govereau P. and Birkedal L. *Ynot: Reasoning with the awkward squad*, ACM SIGPLAN Int. Conf. on Functional Programming, 2008, p. 12.
- [19] OpenSMT project. <http://code.google.com/p/opensmt/>
- [20] Parkinson M. *Local Reasoning for Java*. PhD Thesis, Cambridge University, 2005, 169 p.
- [21] Parr T. *The Definitive ANTLR 4 Reference: Building Domain-Specific Languages*. O'Reilly, 2013, 328p.
- [22] Pavlu V. *Shape-based Alias Analysis. Computing Alias Sets from Shape Graphs to Evaluate the Precision of Shape Analyses*. VDM Verlag Dr. Müller, 2010, 117p.
- [23] Reynolds J. C. *Separation Logic: A Logic for Shared Mutable Data Structures*, Lecture Notes in Computer Science, 2002, pp.55-74.
- [24] Reynolds J. C. *An Introduction to Separation Logic*. Carnegie Mellon University, 2009, 204p.
- [25] SAFECODE within LLVM project. <http://llvm.org>
- [26] Sagiv M., Reps T., Wilhelm R. *Parametric shape analysis via 3-valued logic*. ACM Trans. Program. Lang. Syst., 2002, 24, pp. 217-298.
- [27] Sterling L., Shapiro E. *The Art of Prolog (2nd ed.): Advanced Programming Techniques*, MIT Press, 1994, 552 p.
- [28] Tofte M., Talpin J.-P. *Implementation of the typed call-by-value λ -calculus using a stack of regions*. Proc. of the 21st ACM SIGPLAN-SIGACT. 1994, pp. 188-201.
- [29] Valgrind project. <http://www.valgrind.org>

Energy-Aware Design of Embedded Software through Modelling and Simulation

José Antonio Esparza Isasa^{*}, Peter Gorm Larsen^{*} and Finn Overgaard Hansen[†]

Department of Engineering^{*}, Aarhus School of Engineering[†]

Aarhus University

Aarhus, Denmark

{jaei, pgl}@eng.au.dk^{*}, foh@iha.dk[†]

Abstract—We present a model-driven engineering approach that enables to take energy consumption into account during the development of embedded software. In this approach we address all the constituents of a typical modern embedded solution (mechanics, communication and computation subsystems) through the application of different modelling technologies. This makes it possible to evaluate the implications of different software and system architectures in the system’s energy consumption. Additionally it facilitates the exploration of the design space without having to prototype each candidate solution. We also provide details on the application of this approach to the development of a medical grade compression stocking and the benefits this approach has brought to the project currently developing this system.

I. INTRODUCTION

Modern embedded solutions are typically a combination of computing units, communication interfaces and mechanical subsystems and they can operate both autonomously or as part of a network. This makes embedded solutions heterogeneous systems that are very hard to design [1]. In addition many embedded systems are battery powered and therefore they present the added complexity of being energy efficient while still fulfilling their operational requirements [2].

A possible tool to cope with complexity is the application of abstract modelling. Modelling can be used to represent the system at the highest level of abstraction. These abstract models can be progressively transformed into a concrete system realization [3]. This approach is known as model-driven engineering.

This paper presents a model-driven engineering approach to the design of these complex embedded solutions. This approach makes use of several modelling paradigms in order to represent different aspects of the system and makes special emphasis on the energy performance of different candidate solutions. The modelling activities proposed under this approach are conducted early in the development process and allow design space exploration without requiring physical prototyping. This reduces development time and cost and enables the repeatability of the experimental simulations. Additionally it provides a tool to design quality solutions and to make well-founded design decisions.

The remainder of this article is structured as follows: Section II describes the design approach to energy consumption proposed in this paper. Section III elaborates on HIL and sys-

tem realization following this approach. Section IV describes how this technique is being applied to a case study and its preliminary results. Sections V and VI present future and related work. Finally, Section VII concludes the paper.

II. AN HOLISTIC MODEL-DRIVEN ENGINEERING APPROACH TO EMBEDDED SYSTEMS DESIGN

The approach proposed in here aims at studying the energy consumption in the different subsystems that compose typical embedded solutions. A general representation of such a solution is presented in the SysML block diagram shown in Fig. 1 and its components described below:

- **Embedded Hardware:** represents the electronic hardware that supports the execution of software and possibly other components implementing additional logic in hardware.
- **Embedded Software:** represents the software that controls the operation of the rest of the components in the embedded solution.
- **Mechanics:** represents the mechanical components that are controlled by the system. Interfacing with the mechanical subsystem and the environment from the embedded side is conducted through sensors and actuators.
- **Communications Interface:** represents the communication hardware that makes it possible to establish links with other networked systems.

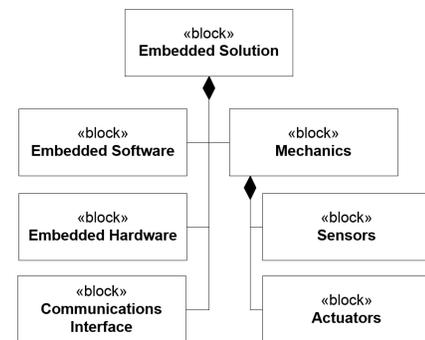


Fig. 1. SysML block representation of a typical embedded solution.

To design an embedded solution comprised by the components presented above so it satisfies its operational requirements is already a challenging task. To design it in

such a way that it is low energy consuming is even more complex. In order to cope with the complexity associated to the design of this kind of systems we proposed the application of model driven engineering techniques that apply System-Level (SL) modelling. SL modelling aims at describing the system under design at the highest level of abstraction and to incorporate progressively system details in order to conduct system analysis. Such a model is gradually transformed into a final implementation. We propose the application of SL design taking into consideration all the energy consuming components in an Embedded Solution in a joint design effort. Additionally we propose Hardware-In-the-Loop as a way to combine executable models with partial system realizations in software and/or hardware running on target. Our approach is said to be holistic because it takes into consideration subsystems that, even though could seem to be unrelated at first sight, they all have an impact in total energy consumption. Therefore, our approach targets the mechanical subsystem, the communication interfaces and the computation logic executed in the embedded software and hardware. All these aspects are addressed in different specific ways in order to obtain energy consumption estimates, that can be used to study the total energy consumption. Additional details on each specific strategy are provided below.

A. Modelling mechanical subsystems

In order to address the design of mechanical subsystems we apply a co-modelling approach in which the engineers use a modelling paradigm able to represent Continuous Time (CT) phenomena and a second one in order to represent Discrete Event (DE) logic [4]. Controlled physical processes (plant) are best represented by using CT abstractions such as differential equations. On the other hand, the control logic that operates the plant is best represented by using logical formalisms.

We proposed a particular way of using this co-modelling approach so it is possible to take into account energy consumption during the design of mechatronic systems [5]. An overview of this approach is presented in Fig. 2. We take as starting point a CT-first methodology [6], in which the modelling starts by focusing on the mechanics of the system and carrying out the control logic modelling afterwards. The CT models are focused on the core functionality that has to be delivered and do not capture heat dissipation due to the conversion efficiency of the electromechanical devices. In case part of the system operation depends on its internal temperature, this could be represented as part of the CT models, since it is a physical process with impact on energy consumption.

Once the system operation has been described, the notion of energy is incorporated into the CT models in a phase that is called models instrumentation. The notion of energy is incorporated only in the CT side since the energy consumption in the DE side is typically negligible if compared with the first one. This instrumentation consist on basically monitoring the variables that have an impact on energy consumption and doing it in a way that does not affect the performance of the

models. After this phase it is possible to co-execute the models and produce a number of energy consumption estimates, that can be used to perform trade-off analysis between the different modelled solutions. In case the energy consumption estimations are fed back to the DE model it is possible to use this approach to model energy-aware system operation, meaning that the system can feature different operational models that are switched among depending on the energy consumed. In principle this approach could be applied by using any tool that supports the co-simulation of DE and CT models.

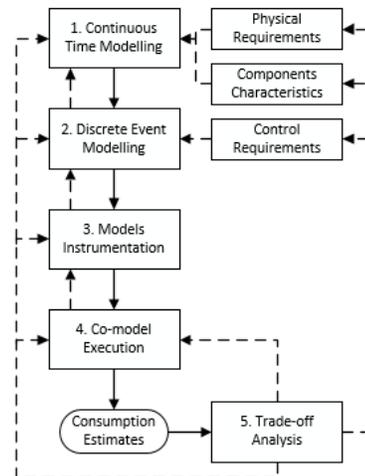


Fig. 2. Overview of the co-simulation based methodology.

In this work we have used the co-execution environment DESTTECS/Crescendo [7], [8]. This environment combines the tools Overture [9] and 20-Sim¹. Overture incorporates an interpreter for the DE modelling language VDM-RT [10]. This language is best suited to represent the control logic supervising the mechanical components and therefore it is used for DE modeling. 20-Sim incorporates a numerical engine that evaluates differential equations. Additionally it supports abstractions built on top of differential equations in the form of bond and block diagrams. This tool is best suited to represent the mechanical side of the system (CT side). The DESTTECS environment synchronizes the co-execution of the models providing a common notion of time for both models. Additionally it allows the specification of a number of controlled and monitored variables between the VDM-RT and the 20-sim simulations so the supervision of the plant is possible. Additional details on how DESTTECS and the process presented above have been applied in a concrete case study will be provided in Section IV.

B. Modelling computation subsystems

Modern microcontrollers can switch between different operational modes in order to reduce energy consumption. These operational modes temporarily disable the CPU and certain peripherals in order to achieve such a reduction. Switching

¹20-Sim official website: <http://www.20sim.com>

between operational modes takes time that might have an impact in the real-time performance of the system under study. An example of different power modes can be seen in Fig. 3. In this case the processor features two low power operational states: Hibernate and Sleep. Sleep allows a fast CPU wake-up based on internally or externally generated events. Hibernate requires additional time to wake-up the CPU and it can react only on external events. However it can lower the energy consumption even further. The CPU controls from software how to switch between these modes.

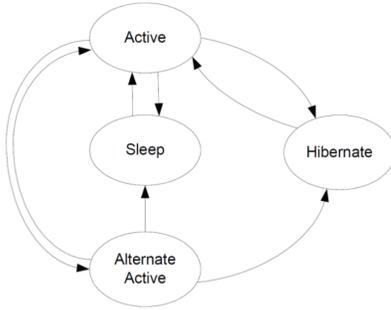


Fig. 3. CPU states in an implementation of a Cortex M3 ARM processor.

In order to explore the application of the different sleep modes, software strategies and architectures required for low power operation we propose the application of the modelling language VDM-RT. This language incorporates the abstraction **CPU** that represents an execution environment in which parts of the model can be deployed. Besides representing the computational support it also incorporates a real-time operating system layer. Logic running in VDM-RT CPUs can represent single or multi-threaded software implementations as well as dedicated hardware blocks depending on how they are configured [11]. However, VDM-RT CPUs do not feature the notion of low power consuming states since they are always active and therefore able to perform computations. An initial approach to overcome this limitation was a design pattern structure that regulated the access to the CPU as a resource by the logic running on it depending on the state of a flag [12].

As a way to overcome this situation we have extended the VDM-RT language by adding two new constructs to manage CPU states: **CPU.sleep()** and **CPU.active()** [13]. The addition of these two new constructs implied the modification of the VDM-RT interpreter and the scheduler built into the Overture platform. In addition to these extensions we proposed specific ways to use the **sleep** and **active** operation so the models can represent accurately the low power operation of real platforms. These templates show how to model a CPU wake-up based on an interrupt triggered by externally generated events and based on internal sleep timers. Model simulation produces a log file that registers in which states the CPU has been operating and for how long. This information together with the electrical characteristics of the CPU under consideration allows to represent the power consumption of the device over time. The integration of this curve over time

results on the total energy consumption on the computational side.

The application of a modelling-based approach to the computational side of the system brings a number of advantages to the design of the software. Besides the obvious case of exploring different sleeping policies for a single CPU without having to prototype them, more complex cases in which several CPUs are involved can be explored. This is especially relevant if the CPUs have to communicate in order to satisfy system requirements.

C. Modelling communication subsystems

The VDM-RT modelling language incorporates the abstraction **BUS** that allows to communicate the **CPU** processing nodes introduced in the previous section. This abstraction can be used to represent point-to-point communication between CPUs in a static way. Communication performed over VDM-RT BUSES is assumed to be error-less, so any kind of communication problems such as information (packet) loss has to be modelled on top. At this point the BUS abstraction does not incorporate any notion of energy consumption during communication.

1) *Modelling network topologies*: We propose the application of a design pattern structure to overcome some of these limitations. Our initial approach takes as an example the communication in a wireless context but it could easily be extended to a different one. In Fig. 4 this structure is presented through a UML diagram. The general idea behind this pattern is to create a star topology network in which each networked embedded system is connected to a central component, that runs a simulated transmission medium. This structure is applied in VDM-RT by using CPUs to represent each networked device as well as a central component simulating the medium. Finally buses are communicating each device model with the medium model. In this way there is no direct connection between the individual CPUs representing the devices and any transmission goes through the simulated wireless medium. By using some of the VDM abstractions one can easily establish relations between the CPUs in the simulated wireless medium to represent whether a communication between two nodes is possible or not. Analyzing these “connection maps” during model execution is especially easy due to the expressiveness of the VDM language and it can be accomplished by using map comprehensions. If model simulation time is a concern the connection maps can be translated into a look-up file in which it is explicitly stated the relation between all the networked elements. This structure solves the initial problem of representing a realistic topology of a small scale embedded network in VDM-RT.

2) *Introducing the notion of energy consumption*: In order to represent the energy consumption we focus on modelling the operational state of the communications interface of the embedded device. We consider operational states the different modes in which the communication interface can be working, typically: transmitting, receiving or deactivated. For each mode the manufacturer provides an average power consumption

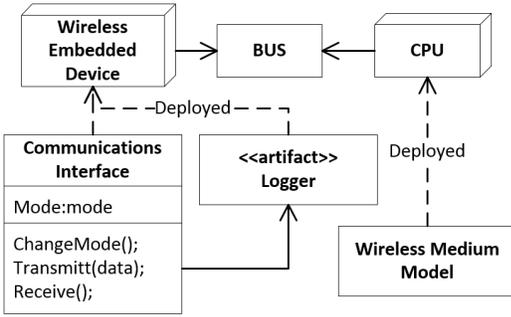


Fig. 4. Design pattern structure to represent wireless communication.

figure that can be used in the VDM-RT models. Changes among these operational modes are logged during model simulation and analyzed when it has been completed. Based on the transitions between the states and for how long the device has stayed on those states one can calculate the evolution of the power consumption over time and hence the total energy consumption during system operation.

Once the notion of energy consumption and the possibility of modelling different network topologies have been facilitated, it is possible to conduct the analysis of communication related problems. Some of these include but are not limited to, routing algorithms, network services, latencies or time synchronization between nodes. All these factors could be analyzed against energy consumption in order to get estimates that would allow an energy aware design of the communication subsystem, including communication software as well as, to some extent, hardware.

One of the advantages of using this structure is the clear separation between the connection map representing the network topology and conditions and the individual networked elements, even though the simulation of both is conducted in the same modelling environment. The main disadvantage of this approach at first sight are the limitations regarding the number of networked elements. We consider this approach valid only for small scale networks. However additional work is necessary to establish its practical limitations.

The approach to communications modelling proposed in this section is conducted only in VDM-RT without involving any other modelling paradigm. A co-simulation approach could be interesting to represent mobile communication nodes or a changes in the environment in which the network is deployed.

III. SYSTEM REALIZATION AND HARDWARE IN THE LOOP

The approach presented above aims at tackling the design problems through modelling and simulation however, at some point, the system has to be realized. Given the fact that a strong emphasis has been placed on the modelling of the system it is desirable that the models created are used as much as possible during the system realization phase. This could include the combination of partial system realizations with models, allowing the co-execution of models with system realizations. The approach that we propose in this work is

exemplified in Fig 5. In this diagram we show an initial VDM model of a system that executes a three phases algorithm in which data is acquired, processed and finally an output is provided.

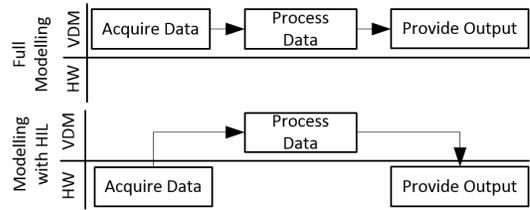


Fig. 5. Overview of the co-simulation based methodology.

We have applied this principle allowing the combination of VDM executable models running in a Workstation with actual components implemented in a Device Under Test (DUT) [14]. These components can be both hardware and software components. An overview of this Hardware In the Loop setup is presented in Fig. 6. In addition to the components mentioned above the system incorporates a Stimuli Provider able to simulate external inputs and a Logic Analyzer able to monitor the evolution of different logical signals. The VDM execution environment is able to interface the Logic Analyzer that can measure the time it takes to execute system realizations running on the DUT. This time figures can be manually incorporated into the VDM model and therefore increasing the fidelity of the model simulation results.

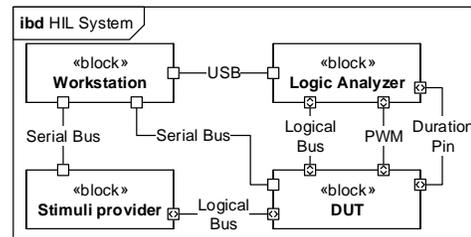


Fig. 6. SysML Internal Block Diagram showing the hardware connections to the DUT.

IV. APPLICATION AND PRELIMINARY RESULTS

The approach proposed in this work is applied to the development of an intelligent compression stocking to treat leg venous insufficiency. This stocking is shown in Fig. 7 and it is composed of: an inner stocking (1), an inflatable stocking responsible for delivering the required compression levels (2), a pneumatical circuit composed of valves, pumps and a manometer (3), and an embedded system implementing the control logic and interfacing hardware and integrating a Bluetooth-based communication interface (4). This portable device is battery-operated and it is required to work for at least 14 hours. A complete description of this device can be found in [15]. As it is explained above this device is

composed of mechanical, computational and communication subsystems that have to be energy efficient so the device autonomy requirements can be met.

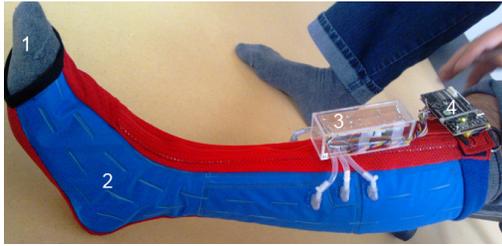


Fig. 7. The medical grade compression stocking.

A. Modelling of the Compression Principle

In order to study the mechanical subsystem we have applied the modelling process presented in Section II-A. The modelling of the mechanical system by itself was already beneficial for the project since it allowed us to gain a thorough understanding of the pneumatics and the physics behind the compression principle. Based on these models we were able to determine that a certain compression strategy was not feasible without having to prototype it. Additionally we were able to model different control software in VDM-RT and co-simulate its performance together with the mechanical models.

The analysis on both control strategies and mechanical pneumatic configurations, provided a number of energy consumption estimates that helped on deciding which system configuration was optimal. These suggestions had an impact on the system realization and introduced improvements on the software that increased its energy efficiency.

B. Modelling of the Embedded Software

We have applied the modelling techniques presented in Section II-B in order to explore two different embedded architectures in a concrete scenario: the regulation. The air pressure level in the air bladders have to be monitored periodically and kept at certain levels so proper compression is delivered to the limb. Through the regulation process the controller reads a manometer, compares the value retrieved against the expected one and depending on this triggers the pump or vents the bladders accordingly. This logic is implemented as a software component and requires the CPU to be active. However and depending on the kind of sensors that are used the CPU can be sleeping for a longer period of time. We have used VDM-RT modelling to study the energy consumption of two different kind of sensor configurations: the first one uses smart sensors that wake up the CPU in case an overpressure event occurs and the second one uses passive sensors that require a poll from the CPU in order to provide a reading. In the first case the CPU presented a lower power consumption than in the second case, since the sensors could run independently from the CPU. In the second case the CPU power consumption was higher because it required periodic wake-ups in order to check the sensors, which were not running independently. These results were

expected since this was a simple case, however the purpose of applying the technique in this case was to show the modelling principle in a simple case study.

The predictions provided by these simulations were confirmed by measurements conducted on system realizations for both architectures with a fidelity of up to 95% [13].

C. Modelling of Communication

The modelling of the communication system remains as future work. A complete overview of the different communication scenarios in which this device can operate is presented in [16]. The intention is to model the critical scenarios in which the device is running on batteries. Based on these models we aim at making energy consumption estimations and evaluating computation vs. communication trade-offs.

V. FUTURE WORK

We are planning to extend the work presented in here so the energy consumption analysis of the communication is also possible. Additionally we are planning to apply the analysis of energy consumption in computation in a more complex situation and possibly combining it with the communication, therefore being able to represent and analyze computation vs. communication trade-offs. We are also aiming at applying some of the modelling techniques presented in here to a second system so it is possible to make a stronger case for the SL energy-aware design approach for embedded solutions.

VI. RELATED WORK

The energy consumption problem in today's embedded solutions is well recognized and one of the top research priorities [2]. System Level design is also a well established technique especially in the hardware world, that it is seeing its expansion to other non-computing domains [3]. However and to our knowledge the application of System Level design with the particular intention of addressing the energy consumption problem during the development process through modelling and prototyping has not been formulated previously. Even though energy consumption in all subsystems has not been addressed in a single design effort previously significant work has been conducted in the individual fields of energy consumption in computation, communication and mechatronic systems.

Regarding energy consumption in computation, extensive work has been carried out in order to characterize different layers of abstraction. Some authors propose very accurate characterization of concrete computing platforms by taking into account energy consumption at the micro-architectural and the instruction level [17], [18]. This differs from the work presented in here in the fact that we consider an average power consumption figure within the active state of the CPU in order to obtain a coarse grained estimation over time. Other authors focus on the characterization of energy consumption at the service level [19]. In this case the authors consider the energy consumption at the OS level when the CPU is active. In our work we consider a single energy consumption figure for all

the services provided by the OS, however, in case some of the OS operations result in a longer time having the CPU active, this will be considered under our approach as well even though the services have not been individually characterized. A more comprehensive review of techniques to study power consumption in computation can be found in [20].

As in the computation case, modelling of communication has been conducted at different levels of abstraction, ranging from energy consumption at the communications interface level to higher layers such as routing or application [21], [22]. Our work makes use of more simple power consumption models that, even though they are based on fixed average power consumption estimates are expected to provide sufficient detail to enable trade-off analysis of network algorithms.

Energy consumption in mechatronic systems is typically addressed as a particular application of well-established modelling platforms such as Matlab, Ptolomy or Modelica. Energy consumption has been typically considered just as any other design factor of industrial grade equipment and mechatronic components have not been traditionally considered together with embedded devices. However this situation is changing due to the increasing relevance of Cyber-Physical Systems [2].

VII. CONCLUSIONS

This paper has presented a modelling approach to energy-aware design of embedded systems. The preliminary application of this approach to a case study has enabled the exploration of different control algorithms, different hardware and software architectures and different mechanical configurations. This has made it possible to evaluate system performance against energy consumption early during the development process without needing a physical prototype. This work will be complemented in the near future with a study of energy consumption from the communication point of view. A more in-depth description of this work can be found in [24].

We hope that the approach proposed can inspire other researchers working with modelling applied to embedded system development and, to some extent, enact the application of modelling in the design of real embedded solutions.

REFERENCES

- [1] Thomas A. Henzinger and Joseph Sifakis, "The Embedded Systems Design Challenge," in *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, 2006, pp. 1–15.
- [2] Banerjee, A. and Venkatasubramanian, K.K. and Mukherjee, T. and Gupta, S. K S, "Ensuring Safety, Security, and Sustainability of Mission-Critical Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 283–299, 2012.
- [3] S. K. Gupta, T. Mukherjee, G. Varsamopoulos, and A. Banerjee, "Research Directions in Energy-Sustainable CyberPhysical Systems," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 57 – 74, 2011.
- [4] J. Fitzgerald, K. Pierce, and P. G. Larsen, *Industry and Research Perspectives on Embedded System Design*. IGI Global, 2014, ch. Collaborative Development of Dependable Cyber-Physical Systems by Co-modelling and Co-simulation.
- [5] J. A. E. Isasa, F. O. Hansen, and P. G. Larsen, "Embedded Systems Energy Consumption Analysis Through Co-modelling and Simulation," in *Proceedings of the International Conference on Modeling and Simulation, ICMS 2013*. World Academy of Science, Engineering and Technology, June 2013.
- [6] J. F. Broenink and Y. Ni, "Model-Driven Robot-Software Design using Integrated Models and Co-Simulation," in *Proceedings of SAMOS XII*, J. McAllister and S. Bhattacharyya, Eds., jul 2012, pp. 339 – 344.
- [7] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. Jovanovic, K. Pierce, and W. F., "Design Support and Tooling for Dependable Embedded Control Software," in *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*. ACM, April 2010, pp. 77–82.
- [8] J. Fitzgerald, P. G. Larsen, and M. Verhoef, Eds., *Collaborative Design for Embedded Systems – Co-modelling and Co-simulation*. Springer, 2014, in press.
- [9] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef, "The Overture Initiative – Integrating Tools for VDM," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, pp. 1–6, January 2010. [Online]. Available: <http://doi.acm.org/10.1145/1668862.1668864>
- [10] K. Lausdahl, P. G. Larsen, and N. Battle, "A Deterministic Interpreter Simulating A Distributed real time system using VDM," in *Proceedings of the 13th international conference on Formal methods and software engineering*, ser. Lecture Notes in Computer Science, S. Qin and Z. Qiu, Eds., vol. 6991. Berlin, Heidelberg: Springer-Verlag, October 2011, pp. 179–194, ISBN 978-3-642-24558-9. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2075089.2075107>
- [11] J. A. E. Isasa, P. G. Larsen, and K. Bjerger, "Supporting the Partitioning Process in Hardware/Software Co-design with VDM-RT," in *Proceedings of the 10th Overture Workshop 2012*, ser. School of Computing Science, Newcastle University, 2012.
- [12] J. A. E. Isasa and P. G. Larsen, "Modelling Different CPU Power States in VDM-RT," in *Proceedings of the 11th Overture Workshop 2013*, ser. Aarhus University, June 2013.
- [13] J. A. E. Isasa, P. W. Jørgensen, and C. Ballegaard, "Modelling Energy Consumption in Embedded Systems with VDM-RT," in *Proceedings of the 4th International ABZ conference.*, July 2014.
- [14] J. A. E. Isasa, P. W. Jørgensen, and P. G. Larsen, "Hardware In the Loop for VDM-Real Time Modelling of Embedded Systems," in *MODELWARD 2014, Second International Conference on Model-Driven Engineering and Software Development*, January 2014.
- [15] T. F. Jensen, F. O. Hansen, and J. A. E. et al., "ICT-Enabled Medical Compression Stocking for Treatment of Leg-Venous Insufficiency," in *International Conference on Biomedical Electronics and Devices (BIODEVICES 2014)*, March 2014.
- [16] F. O. Hansen, T. F. Jensen, and J. A. Esparza, "Distributed ICT Architecture for Developing, Configuring and Monitoring Mobile Embedded Healthcare Systems," in *International Conference on Health Informatics (HEALTHINF 2014)*, March 2014.
- [17] Mostafa E.A. Ibrahim and Markus Rupp and Hossam A. H. Fahmy, "A Precise High-Level Power Consumption Model for Embedded Systems Software," *EURASIP Journal on Embedded Systems*, vol. Volume 2011, no. 1, January 2011.
- [18] Sheayun Lee and Andreas Ermedahl and Sang Lyul Min, "An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors," 2001.
- [19] B. Ouni, C. Belleudy, and E. Senn, "Accurate energy characterization of os services in embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2012, no. 1, p. 6, 2012.
- [20] Unsal, O.S. and Koren, I., "System-Level Power-aware Design Techniques in Real-Time Systems," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1055–1069, 2003.
- [21] E. Shih, S.-H. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan, "Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks," in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '01. New York, NY, USA: ACM, 2001, pp. 272–287.
- [22] R. Min and A. Chandrakasan, "A framework for energy-scalable communication in high-density wireless networks," in *Proceedings of the 2002 International Symposium on Low Power Electronics and Design*, ser. ISLPED '02. New York, NY, USA: ACM, 2002, pp. 36–41.
- [23] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava, "Energy-Aware Wireless Microsensor Networks," *signa Processing Magazine*, vol. 19, no. 2, pp. 40–50, March 2002.
- [24] J. A. E. Isasa, "System-Level Energy Aware Design of Cyber-Physical Systems," Department of Engineering, Aarhus University, Finlandsgade 22, Aarhus N, 8200, Tech. Rep., October 2013, available on-line at http://eng.au.dk/fileadmin/DJF/ENG/PDF-filer/Tekniske_rapporter/ECE-TC-16-samlet.pdf.

Energy Aware Congestion Management in Dynamic Wireless Mesh Network

S.P.Shiva Prakash

Research Scholar

JSS Research Foundation

Mysore, Karnataka, India

Email: shivasp26@gmail.com

T.N.Nagabhushan

JSS Academy of Technical Education

Noida, India

Email: tnnagabhushan@gmail.com

Kirill Krinkin, Olga Sholokhova

Saint-Petersburg Electrotechnical University "LETI"

Saint Petersburg, Russia

Kirill.krinkin@fruct.org, sholokhova.olya@gmail.com

Abstract—Wireless Mesh Network(WMN) has emerged as most widely used popular network due to its self discovering, self organizing and self healing characteristics. However, WMN based on Time Division Multiple Access(TDMA) Medium Access Control(MAC) protocol suffer from poor Quality of Service(QoS) due to low throughput and high network overhead. Using different rate control mechanisms in such networks can further increase problems during mobility of nodes. Efficient slot allocation and scheduling is needed in congestion control algorithms to improve the performance of such networks. Towards this context, IEEE 802.11s standard introduced a mechanism called local congestion monitoring (LCM) that considers the amount of incoming and outgoing traffic to control congestion for a single hop network. Further many researchers have proposed algorithms to control congestion keeping nodes as static. The drawback of existing protocols is that it neglects the importance of energy resource during slot allocation which plays a major role in network performance. Also considering mobility of a node which results in high congestion due to frequent link breakages and high energy consumption due to re-establishment of route during routing process. Hence to overcome the problem, In this work we propose a novel congestion control protocol called Energy aware congestion management protocol (EDMA) in TDMA MAC under mobility. The proposed model consists of four modules namely, Energy Aware Routing, Node Position Identifier, Energy Evaluator and Congestion Control TDMA slot allocator. The slot allocation in TDMA MAC works in two sub phases: Energy-aware Slot Allocator and Energy-aware Slot Scheduler that allocates and schedules slot in TDMA MAC considering both energy status and mobility metrics. The working of proposed model is presented and analysis shows that proposed model results in congestion free network thus improving QoS in WMN.

I. INTRODUCTION

In WMN each mobile host acts as a router and helps to route the information for a desired host. It supports peer to peer communication and easy to deploy. Congestion refers to a network state where a node or link carries loads of data that it may deteriorate network service quality and resulting in queuing delay, frame or data packet loss and the blocking of new connections. IEEE 802.11s standard introduced a mechanism called local congestion monitoring considering based on amount of incoming and outgoing traffic to control congestion for a single hop network [11].

In general, congestion is a number of packets being sent through the network is greater than the number of packets the network can handle. Figure 1 depicts occurrence of congestion in wireless mesh network due to limitation of queue size and

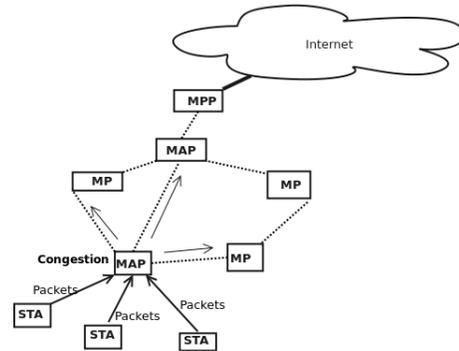


Fig. 1: Congestion in network

incoming and outgoing capacity of a data rate and also lack of awareness of energy resource at a node during slot allocation. In wireless mesh network nodes can move independently with different velocity and pause time. A mobility of a node results in high energy consumption due to the link breakages considering relative mobility between two nodes, this results in link failure during mobility of node as node changes its position and results in node being out of transmission connectivity range. Congestion is basically caused at two layers:

- MAC layer
- Network layer

At MAC layer during the allocation of data slots to the nodes i.e., when two nodes simultaneously access the same data slots. Here in above fig node n1 and n2 are requesting

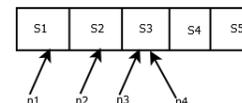


Fig. 2: Collision leads to Congestion

slots s1 and s2 for communication and slots are allocated successfully to node n1 and node n2 but at next slot allocation time two node i.e. n3 and n4 are accessing the same slot for communication this leads to collision.

At Network layer congestion occurs when the incoming link data rate is greater than outgoing link which results in long

delay. Consider figure 3 as an example, source is transmitting the packets to destination via relay node 3,5,and 7 using the data rate at 6 Mbps but the relay node 3 can provide the data rate of 3 Mbps. Hence, the data packet needs to wait in queue for longer time; when the capacity of buffer exceeds, next incoming packets are discarded; there is a possibility for unexpected delay or packet loss in transmission. In WMN, congestion may occur in any intermediate node due to limited resources, when data packets are being transmitted from source to the destination.

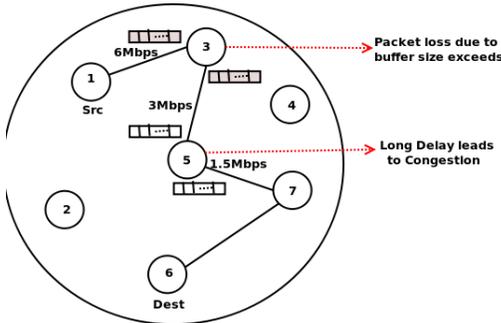


Fig. 3: Collision leads to Congestion

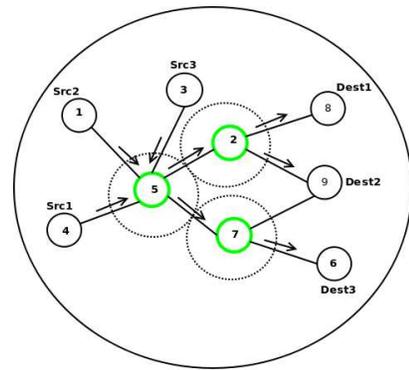
The mobility of nodes is also a major role within WMN's due to limited transmission range; this can cause link failure that lead to recalculate their routing information; this consumes processing time as well as battery power.

In general, packet loss occurs due to congestion where as in WMN, packet loss may occur due to node mobility too.

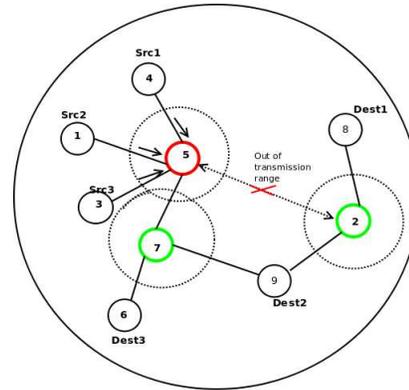
Figure 4(a). shows that there is transmission occurs within range and successful packet transmission. Three source nodes are requesting at same relay node5. The communication range between node5 and node2 are within range and node5 and node7 too at t_0 . Since nodes are within transmission range, the packets successfully forwarded but at the time t_1 in Figure 4(b). the transmission range between node5 and node2 are exceeds the range threshold value due to mobility this lead to link breakage and caused packet loss at node5 and node2. This may lead to high energy consumption due.

In conventional congestion control mechanism installed with TDMA MAC, slots will be allocated to the requested node based on the arrival time of nodes. This works well in static nature of a node as no frequent change in slot allocation is required. It may results in congestion due to the unaware of energy source at a node. Hence, there is a need to consider the dynamic nature and energy resource of a node during slot allocation in TDMA MAC protocol.

The rest of the paper is organized as follows: Section 2 define the problem statement and Section 3 discuss related previous works. Section 4 presents the proposed model and Section 5 define the mathematical model while Section 6 depict the algorithm for proposed model. Result and observations are discussed in Section 7. Section 8 presents the Conclusion and scope for future work.



(a) Transmission occurs within range



(b) Link breakage

Fig. 4: Change in position of the nodes due to mobility, causing the link breakage

II. RELATED WORKS

In this section we present the related works carried out by various researchers to control congestion in WMN.

Focusing on the scheduling of a single multicast session where each receiver gets the same throughput, authors [5] proposed an interference-aware fair scheduling algorithm named LOF for multicast in wireless mesh network. It guarantees that different receivers of a single session get the same throughput. Authors of [6] have proposed the Bellman-Ford TDMA scheduling algorithm. It takes the scheduling delay into consideration while taking advantages of spatial reuse. Authors of [7] have discussed about the benefits of a centralized implementation of the NUM based rate allocation. Authors of [8] proposed a model in order to avoid secondary queuing delay (i.e delay occurred in multi hop network when two or more nodes requesting for the same channel) for multiple hops in a network. A separate slots will be allocated based on the arrival time so that further communication can takes place through same allocated slot. Authors of [9] have proposed an algorithm for the draft standard outlines an optional hop-by-hop congestion control mechanism. Each MP observes the level of congestion based on the amount of incoming and outgoing traffic (local congestion monitoring). When the traffic increases to a point

such that the MP is unable to forward and source data upstream as fast as the incoming rate, congestion occurs, and the MP must notify one-hop neighbors (local congestion control signalling). These neighbors respond by limiting the rate at which they are sending to the congested MP (local rate control). Authors of [4] have proposed a mechanism in which nodes in the network are first labelled either even or odd. Then, while determining paths, only those paths that go through nodes having alternate labelling are considered. Using sub-channelization of OFDMA, secondary interference between two links in the same slot is prevented by assigning different channels to the links. Once the slot requirements and routing paths are determined, each node employs a local (wireline-type) scheduling policy. The scheduling policy determines the order in which packets leave the buffer at each node, and the authors show that such a mechanism provides two-approximation bounds for the end-to-end delay. The authors of [10] have proposed a routing heuristic model in which first, the shortest-hop algorithm is used to determine a path. If one or more edges on the path are blocked, those are removed from the graph and the heuristic is applied again to find a suitable path. In the literature [3], [1], and [2] several TDMA broadcast scheduling algorithms can be found. These algorithms use several heuristic approaches, graph coloring, in order to solve the Broadcast Scheduling Problem (BSP). The optimum frame length is obtained and the network throughput maximization is achieved in all the previous algorithms.

Most of the related works have focused on controlling congestion keeping nodes in static position. However, nodes in mobility are proven to undergo several issues in connection with congestion due to frequent change in direction. As a result it reconstructs route which may further lead to more energy consumption. Hence congestion may occur due to lack of sufficient energy at relay node to process the rested node especially during mobility. This also causes delay in forwarding packets from relay node after the allocation of slot to the requested nodes. So we propose a protocol that manages congestion control considering both energy constraints and mobility metrics.

III. PROBLEM STATEMENT

Consider a network of n nodes where n_1, n_2, n_3 are requesting nodes. Let R be the relay node. In conventional TDMA based slot allocation, the slot will be allocated based on the arrival time of a request. Also, considering the dynamic nature of a node due to the mobility there are possibility of link failure which results in route re-establishment, this may also results in high energy consumption compared to static nature of a node. This may lead to packet loss due to the lack of sufficient energy at a node n to perform transmission operation such as packet forwarding at a particular interval of time and results in congestion of a network. Hence, there is a need to define a protocol which considers the energy status of a node during slot allocation in TDMA which results in congestion free network.

Factors affecting congestion

- 1) Packet arrival rate exceeds the outgoing link capacity

- 2) Buffer size exceeds to store arriving packets
- 3) Degradation battery below the pre-defined threshold
- 4) Link breakage due to mobility

IV. PROPOSED MODEL

In this section we present our proposed model to control congestion. The proposed EDMA model is shown in Figure 5.

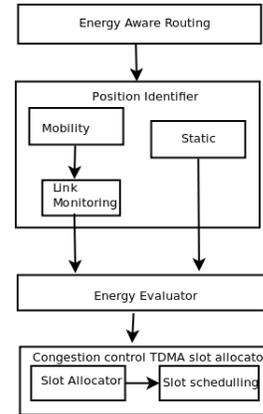


Fig. 5: Proposed model

The proposed model consists of four phases namely,

1. Energy Aware Routing
2. Node Position Identifier
3. Energy Evaluator
4. Congestion Control TDMA slot allocator

1. Energy Aware Routing: The proposed model begins by routing, this is a initial phase and routing of packets takes place based on the energy criteria. When source broadcast the message to its neighbor nodes to reach the destination. Multiple reply from the neighbor nodes will be generated. The selection process of relay node takes place based on energy i.e., required energy threshold will be defined for relay node. The remaining energy of the reply nodes must be greater than or equal to the defined threshold and the nodes having highest energy will be selected. Each relay node that is selected acts as a source node and the same process takes place until its reach to the destination node.

2. Node Position Identifier: It identifies the position of a node based on static or dynamic nature. In static, packet size and data rate used as parameters to calculate the static remaining energy of a node. In dynamic, Random Way Point model is used as a mobility model considering velocity, pause time, packet size and data rate as a parameters to calculate the dynamic remaining energy of a node.

3. Energy Evaluator: It calculates the energy consumption of a node considering the parameters in static or dynamic nature. It sets the required energy threshold for a requested node. It also calculates the remaining energy of a node at a regular interval of time.

4. Congestion Control TDMA slot allocator: It has two sub phases such as,

A.Slot allocator : It allocates the TDMA slots of a node by comparing the required energy of a requested node and the

remaining energy of its own node. The slot will be allocated if the remaining energy of a node is greater than the requested node energy threshold. Otherwise, it will be rejected.

B.Slot Scheduling : After Slot allocation within a node, the requested node will be scheduled in TDMA based on velocity and pause time. The node with minimum velocity and pause time is given higher priority during slot scheduling.

A. Packet formats used

In this section we present the different packet formats used.

Routing Table Format:

Source ID 1	Source ID 2	Next Hop	Destination ID 1	Destination ID 2	Energy	Slot Allocation ID
-------------	-------------	----------	------------------	------------------	--------	--------------------

Fig. 6: Routing Table

Where,

SourceID: Address of Source node(constant).
 OriginatID: Address of Originating source node(changeable).
 Next Hop: Address of next neighbor node.
 DestinationID: Address of Destination.
 Energy: Energy of corresponding node.
 Slot Allocation ID: ID of the allocated slot.

RREQ(Route Request) Packet Format :

Origin ID	Next Hop
-----------	----------

Fig. 7: RREQ Format

Where,

OriginID: Address of originator.
 Next Hop: Address of next neighbor node.

RREP(Route Replay)Packet Format:

Destination ID 1	Originator ID	Hop Count
------------------	---------------	-----------

Fig. 8: RREP Format

Where,

DestinationID: Address of Destination.
 OriginatorID: Address of originator.
 Hop Count: Total number of hop count.

Slot Allocator Format:

Node ID	Slot Allocation ID
---------	--------------------

Fig. 9: Slot Allocator Format

Where,

Node ID: Address of the node.
 Slot Allocation ID: Address of the allocated slot.

V. ALGORITHMS

In this section we present the algorithms used in our proposed model.

Algorithm 1 Energy Determiner

Set $InitialEnergy = 10J$
 $Ec(ps, dr) = N \sum (Ec_t x, Ec_r x, Ec_r el, Ec_i dle)$
 $Ec_i \& = \frac{I \times V \times T_p \times P_s}{dr}$
 $RE = InitialEnergy - Ec_i$

Where,

I: Current
 $I = 330 \text{ mA}$ (for t_x)
 $I = 220 \text{ mA}$ (for r_x)
 V: voltage ($V=5$)
 T_p : time taken to transmit the packet p
 P_s : Packet size
 dr: Data rate in Mbps

The Algorithm 1 calculates the energy consumption of a node and set the initial energy for all the nodes in a network. Remaining energy will be calculated based on the transmission modes with help of data rate and packet size in static nature of a node where as in dynamic it considers data rate, packet size and also velocity and pause time. It also sets the required threshold.

Algorithm 2 Slot allocation

for each node N **do**
 Call Energy Determiner
if $RE \geq ReqE$ **then**
 Slot Allocated
end if
end for

Where,

N: Requested nodes
 ReqE: Required Energy
 RE: Remaining Energy

Algorithm 2 allocates the TDMA slots of a node by comparing the required energy of a requested node and the remaining energy of its own node. The slot will be allocated if the remaining energy of a node is greater than the requested node energy threshold. Otherwise, it will be rejected.

Algorithm 3 Slot Scheduler

for each node N **do**
 $SlotAllocator = \min(V, P)$
end for

Where,

N: Requested nodes
V: Velocity
P: Pause time

The slot scheduler algorithm 3 is executed after algorithm 2 with in a slot allocated node. It schedules slot for the requested node in TDMA based on velocity and pause time. The node with minimum velocity and pause time is given higher priority during slot scheduling.

Algorithm 4 Energy-aware Congestion-management Protocol

```

for each node  $N$  do
  Broadcast RREQ message
  Send RREP message
  Call Energy Determiner
  Call ESS
end for
if path exists then
  choose the path based on energy parameter with minimum hop
  packet transmission
  if link breakage then
    send RERR to source
    Re- discover the path
  end if
end if

```

Algorithm 3 depicts the energy aware congestion management algorithm in which a broadcast RREQ message is sent by source nodes to its neighbor nodes during routing. In congestion control mechanism relay nodes plays a major role when there is multiple request to forward the packets. Hence relay node utilizes the Energy Determiner algorithm to calculate the remaining energy and set the required energy threshold value to allocate the slot for a requested node.

VI. ANALYSIS OF PROPOSED EDMA MODEL

Consider a network of 10 nodes as an example. Each node maintains an array of 5 slots in TDMA MAC. Let S1,S2,S3 be the source nodes and D1,D2,D3 be the destination nodes respectively. R1,R2 and R3 be the Relay nodes. Let a packet of size P with data rate D_r is sent from S1,S2 and S3 to D1,D2 and D3 respectively. The source nodes s1 move with a velocity=15, pause time=10, s2 move with a velocity 10 and a pause time 5 and S3 move with a velocity=5, pause time = 0. Energy consumption for single packet is 0.12 at source node and 0.18 at relay node and also energy consumption for a RREQ and RREP message is 0.05J. Figure 10 depicts the example network.

TABLE I: Analysis Table

Node	RequiredThreshold
S1	5J
S2	5.5J
S3	1.35J

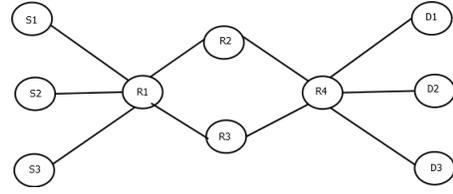


Fig. 10: Initial network

TABLE II: Initial Routing Table

Node	SrcID	OriginID	NextHop	DestID	Energy	SlotAllID
s1		s1			10	
s2		s2			10	
s3		s3			10	
R1		R1			25	
R2		R2			25	
R3		R3			25	
R4		R4			25	

The Table II depicts the initial routing table for all nodes in a given network and its remaining energy before transmission takes place.

A. Conventional TDMA

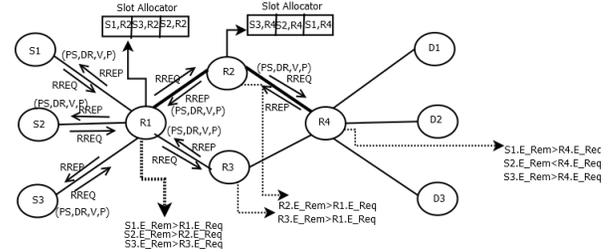


Fig. 11: Conventional TDMA Allocation

Iteration 1: Figure 11 shows the slot allocation mechanism in conventional TDMA. A slot will be allocated for each requested source node at node relay 1 based on the request arrival time and packet will be forwarded to relay R2 node which further allocates the slots to forward the packet to the destination node. Since at the beginning each relay node will have a sufficient energy to forward the packet the packet will be successfully transmitted to the destination.

TABLE III: Routing Table

Node	SrcID	OriginID	NextHop	DestID	Energy	SlotAllID
s1	s1	s1	R1	D1	9.9	
s2	s2	s2	R1	D2	9.9	
s3	s3	s3	R1	D3	9.9	
R1	s1,s2,s3	R1	R2,R3	D1,D2,D3	24.6	s1,s3,s2
R2	s1,s2,s3	R2	R4	D1,D2,D3	24.8	s3,s2,s1
R3	s1,s2,s3	R3	R4	D1,D2,D3	24.9	

The table III shows energy consumption for RREQ and RREP message at nodes. Nodes satisfied Required threshold value and allocated data slot for requested node based on the arrival time.

Source node s1 wants to communicate with destination D1 through its neighbor R1 and it requires 5J of energy to packet transmission.

At source node S2, it wishes to communicate with destination D2 with 10J of energy required and source node S3 requires 5.5J to transmits the packets to its destination D3.

Since allocation of data slots takes place at the relay nodes due to mobility, slot allocation id field of the source node S1, S2 and S3 contains null value.

At Relay node R1, it contains the source id S1,S2 and S3. The data slot for S1, S2 and S3 allocated according to its arrival time and R1 forwards the slot requests to its nexthop R2s.

Relay node R2, contains the source id as S1, S2 and S3, based on the arrival time R2 allocates the slots for source nodes and it has 25J of energy. At relay node R3, same process is continued.

Relay node R4, it contains the source id S1,S2 and S3. It has 25J and its neighbor node are the destination D1, D2 and D3. Allocation of data slots takes place based on the arrival time.

Iteration 2: In routing table IV, source node S1 wishes to send 35 packets, S2 has 25 packets and S3 has 30 packets. All Packets will be transmitted successfully to the relay node R1 and also to relay node R2. We can notice that R2 has allocated slots for S1,S2 and S3.

TABLE IV: Routing Table

Node	SrcID	OriginID	Nexthop	DestID	Energy	SlotAllID
s1	s1	s1	R1	D1	5.7	
s2	s2	s2	R1	D2	6.9	
s3	s3	s3	R1	D3	6.3	
R1	s1,s2,s3	R1	R2,R3	D1,D2,D3	8.4	s1,s3,s2
R2	s1,s2,s3	R2	R4	D1,D2,D3	8.4	s3,s2,s1
R3	s1,s2,s3	R3	R4	D1,D2,D3	24.9	
R4	s1,s2,s3	R4	D1,D2,D3	D1,D2,D3	8.4	s3,s2,s1

Iteration 3: Figure 12shows the slot allocation of TDMA at relay node R1 and R2 respectively. Node S1 energy exceeds required threshold value for next 10 packets. Since slot S1 does not have sufficient energy to transmit packet to relay R4 the packet get dropped. This results in low packet delivery ratio and it will result in congestion.

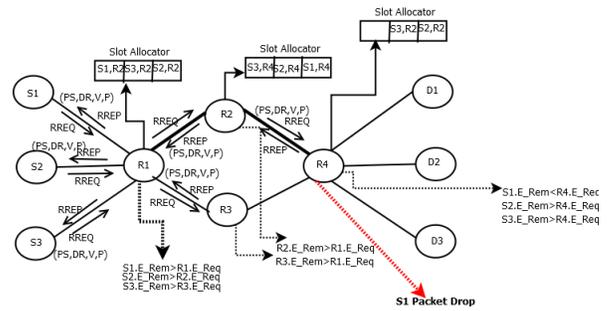


Fig. 12: packet drop at R4 due to insufficient energy

TABLE V: Routing Table

Node	SrcID	OriginID	Nexthop	DestID	Energy	SlotAllID
s1	s1	s1	R1	D1	4.5*	
s2	s2	s2	R1	D2	5.7	
s3	s3	s3	R1	D3	5.1	
R1	s1,s2,s3	R1	R2,R3	D1,D2,D3	6.6	s1,s3,s2
R2	s1,s2,s3	R2	R4	D1,D2,D3	6.6	s3,s2,s1
R3	s1,s2,s3	R3	R4	D1,D2,D3	6.6	
R4	s1,s2,s3	R4	D1,D2,D3	D1,D2,D3	6.6	s3,s2

*s1 exceeds its threshold value

Iteration n: Figure 13 shows the slot allocation of TDMA at relay node R1 , R2 and R4 respectively. We can notice that R2 has allocated slots for S1,S2 and S3. Since slot S1 does not have sufficient energy to forward packet to relay R2 the packets get dropped and also slot S2 does not have sufficient energy to forward packet to relay R2 the packets get dropped after 2 packets have sent. Both of S1 and S2 packets are dropped this results in low packet delivery ratio and it will result in congestion.

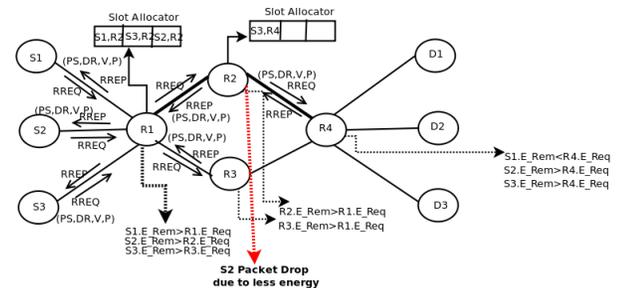


Fig. 13: Packet drop at R4 due to insufficient energy

TABLE VI: Routing Table

Node	SrcID	OriginID	NextHop	DestID	Energy	SlotAllID
s1	s1	s1	R1	D1	4.5 ^{**}	
s2	s2	s2	R1	D2	5.46 ^{**}	
s3	s3	s3	R1	D3	7.03	
R1	s1,s2,s3	R1	R2,R3	D1,D2,D3	6.6	s1,s3,s2
R2	s1,s2,s3	R2	R4	D1,D2,D3	5.88	s3
R4	s1,s2,s3	R3	R4	D1,D2,D3	5.88	s3,s2

*s1 exceeds its threshold value, **s2 exceeds its threshold value

Relay node R1 allocated slots for s1,s2 and s3 but at node R2 the s2 energy degraded below threshold. Hence, R2 allocate slots only for s3 and s1. Slot allocation for s2 packets are get dropped due to insufficient energy this leads to congestion.

B. Proposed MODEL

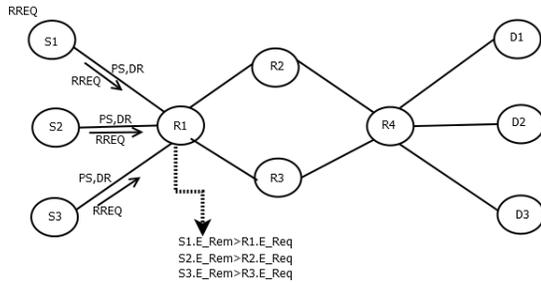


Fig. 14: Broadcasting request message

Iteration 1: Figure 14 depicts the proposed model which allocate the slots for each source node in relay R1 based on the required energy threshold parameter. Firstly the source nodes S1,S2 and S3 broadcasts the RREQ to the one hop neighbor node.

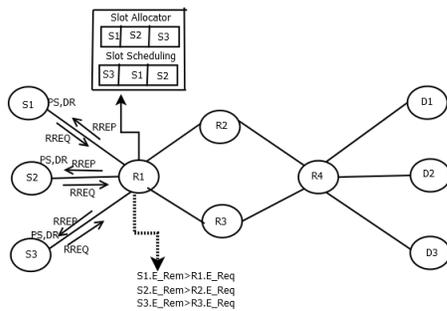


Fig. 15: Slot allocation at relay node R1

TABLE VII: Routing Table

Node	SrcID	OriginID	NextHop	DestID	Energy	SlotAllID
s1	s1	s1	R1	D1	9.9	
s2	s2	s2	R1	D2	9.9	
s3	s3	s3	R1	D3	9.9	
R1	s1,s2,s3	R1	R2,R3	D1,D2,D3	24.6	s1,s2,s3

The slot allocation of node is depicted in VIII

TABLE VIII: Slot Allocation

s1	s2	s3
----	----	----

The slot scheduling of node is depicted in IX

TABLE IX: Slot Scheduling

s3	s1	s2
----	----	----

Iteration 2: Figure 15 depicts the proposed model which allocate the slots for each source node in relay R1 based on the required energy threshold parameter. It calculates the remaining energy of the relay node and the source node. Since the remaining energy of a relay node is greater than the requesting source node it allocates the slot. Further it schedules the slots based on the minimum pause time and velocity. Since source S3 has minimum velocity slot 1 will be allocated to it and Slot 2 is allocated to source S1 and slot 3 will be allocated to source S2.

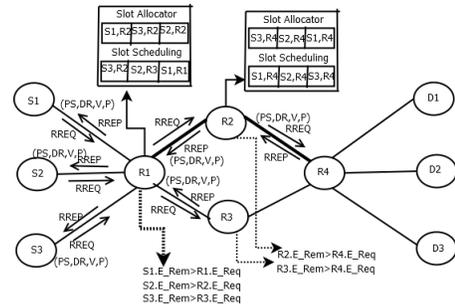


Fig. 16: Slot allocation at relay node R2

TABLE X: Routing Table

Node	SrcID	OriginID	NextHop	DestID	Energy	SlotAllID
s1	s1	s1	R1	D1	9.64	
s2	s2	s2	R1	D2	8.05	
s3	s3	s3	R1	D3	8.58	
R1	s1,s2,s3	R1	R2,R3	D1,D2,D3	23.21	s1,s3,s2
R2	s1,s2,s3	R2	R4	D1	23.21	s3,s2,s1

The slot allocation of node is depicted in XI

TABLE XI: Slot Allocation

s3	s2	s1
----	----	----

The slot scheduling of node is depicted in XII

TABLE XII: Slot Scheduling

s1	s2	s3
----	----	----

Iteration 3: Figure 16 shows the allocation of the slots for each source node in relay R2 based on the required energy threshold parameter. It calculates the remaining energy of the relay node and the source node. Since the remaining energy of a relay node is greater than the requesting source node it allocates the slot. Further it schedules the slots based on the minimum pause time and velocity. Since source S1 has minimum velocity slot 1 will be allocated to it and Slot 2 is allocated to source S2 and slot 3 will be allocated to source S3.

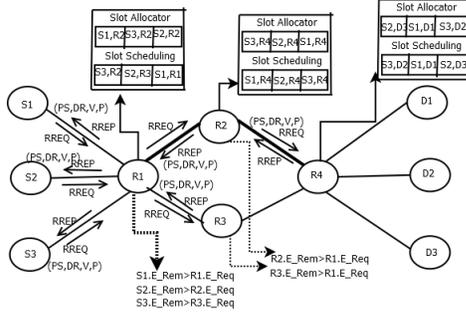


Fig. 17: Slot allocation at relay node R4 and successful transmission

Figure 17 shows the allocation of the slots for each source node in relay R4 based on the required energy threshold parameter. It calculates the remaining energy of the relay node and the source node. Since the remaining energy of a relay node is greater than the requesting source node it allocates the slot and also this results in successful transmission of packets. The routing table of node R4 is depicted in Table XIII.

TABLE XIII: Node R4 Routing Table

SrcID	OriginID	Nexthop	DestID	Energy	SlotAllID
S1,S2,S3	R4	D1,D2,D3	D1,D2,D3	23.21	S2,S1,S3

The slot allocation of node is depicted in XIV

TABLE XIV: Slot Allocation

s2	s1	s3
----	----	----

The slot scheduling of node is depicted in XV

TABLE XV: Slot Scheduling

s3	s2	s1
----	----	----

Figure XIV shows the allocation of the slots for each source node in relay R4 based on the required energy threshold parameter. It calculates the remaining energy of the relay node and the source node. Since the remaining energy of a relay node is greater than the requesting source node it allocates the slot and also this results in successful transmission of packets.

VII. CONCLUSION FUTURE WORK

In this work we have presented a congestion control model that schedules the TDMA MAC slot based on the remaining energy and velocity, pause time during mobility of nodes. Since it considers the important energy resource of a node during the allocation of slot to relay node based on the incoming packet rate and amount of energy required to it, It guarantee the improvement in throughput of a network in mobility. The working of proposed model shows that proposed model results in congestion free network and 100% packet delivery ratio which further improves QoS in WMN. The model will be implemented, tested in network simulator NS3 and behaviour of network will observed in future.

REFERENCES

- [1] Yeo, J., Lee, H., and Kim S, *An efficient broadcast scheduling algorithm for TDMA ad-hoc networks*. Computer and Operations Research, 29(13), pp. 17931806, 2002.
- [2] Vergados, D. J., Vergados, D. D. and Douligeris, C. *A new approach for TDMA scheduling in ad-hoc networks*, In Proc. 10th IFIP international conference on personal wireless communications (PWC05), Colmar, France, August, pp. 279286, 2005
- [3] Vergados, D. D., Vergados, D. J., Douligeris, C. and Tombros, S.L., *oS-aware TDMA for end-to-end traffic scheduling in ad-hoc networks*, IEEE Wireless Communications, 13(5), pp. 6874. 2006
- [4] G Narlikar, G Wilfong, L Zhang, *Designing multihop wireless backbone networks with delay guarantees*, In Proc. INFOCOM. Catalunya, SPAIN, pp. 112, 2006.
- [5] D. Koutsonikolas, S. M. Das, and Y. C. Hu, *An interference-aware fair scheduling for multicast in wireless mesh networks*, Journal of Parallel and Distributed Computing, 2007.
- [6] Petar Djukic and Shahrokh Valaee, *Link scheduling for minimum delay in spatial re-use TDMA*, In Proc. 26th IEEE International Conference on Computer Communications, INFOCOM, pp. 28-36, 2007.
- [7] B. Wang and M. Mutka, *QoS-aware fair rate allocation in wireless mesh networks*, Computer Communications (Special Issue: Resource Management and routing in Wireless Mesh Networks), vol. 31, no. 7, 2008
- [8] Jae-Hyun Kim, Jae-Ryong Cha and Han-Joon Park, *New delay-efficient TDMA-based distributed schedule in wireless mesh networks*, EURASIP Journal on Wireless Communications and Networking, 2012. 2012:369.
- [9] Joseph D. Camp and Edward W. Knightly, *The IEEE 802.11s Extended Service Set Mesh Networking Standard*, IEEE Communications Magazine, 2008
- [10] A Sahoo, P Goyal, *A scheduling and call admission control algorithm for wimax mesh network with strict qos guarantee* in Proc. COMSNETS, Bangalore, India, pp. 2029,2010.
- [11] IEEE, "Draft amendment: ESS mesh networking," *IEEE P802.11s Draft 12.00*, Jun 2011.

An architecture of effective discrete-event simulation engine for early validation of avionics systems

Denis Buzdalov

Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
Email: buzdalov@ispras.ru

Abstract—Nowadays models which are used in the avionics (aviation electronics) development are large and can contain complex behaviour specifications, especially on early stages. This leads to high requirements to simulators which are used for such models analyses. Existing open-source simulators are not applicable or not effective in application on such models. An architecture of a discrete-event simulator using *continuations* approach for avionics models analysis is suggested.

Keywords—*discrete-event simulation, avionics, early validation*

I. INTRODUCTION

An area of creation and verification of avionics (aviation electronics) and other responsible systems is considered.

Nowadays design processes of such systems cannot be performed without modelling techniques usage. Such models are used not only for creation of systems but also for analysis and validation of them (including early validation). Such analyses are usually performed in an automated way because of the great size of analyzed models. That is why we consider only automated ways of the model analysis.

There are a lot of kinds of analysis. One of division aspects of functional characteristics analyses is a division to static and dynamic analyses. Both types are important but each one has features that doesn't allow a type to oust another one.

So, static analyses usually give guaranteed estimation of model characteristics. Because such estimations are given for the model in general, they usually are pretty pessimistic. Also, new type of estimated characteristic often leads developing of new static analysis method.

Dynamic analyses are used to determine more optimistic estimations. But important feature of such methods is that such estimations are determined only for particular cases. This means that estimated characteristics are not guaranteed to be the same in different environments (input data and non-determinism resolution). This does not allow to use dynamic analysis to reason about a model in general.

A lot of kinds of dynamic analysis exist. They can differ in data, prerequisites and aspects that system can be analyzed with.

Simulation is a widely used type of dynamic analysis. This approach allows to estimate both timing and functional properties of designed systems in different cases.

One of particular kind of simulation — *discrete-event simulation* — is considered in this work. This kind of simulation is naturally suitable for the computer systems modelling. In this approach the work of the modelled system is represented as a sequence of discrete events. Each discrete event is an atomic action of internal state change and interaction with outer world and other components. All actions of a single discrete event are performed in a single moment of the simulation time.

This paper considers a problem of having or building of an effective and convenient simulation system for avionics systems analysis.

II. SIMULATION SYSTEM REQUIREMENTS ANALYSIS

There are a lot of ways *how* discrete-event simulation can be performed. They can differ in both which characteristics are taken into account and which input data is required for the simulation system.

A. Events source

One of differences between the discrete-event simulation approaches is in the events source. Some of them consider only *periodic* events which are caused by the time. Also *sporadic* events can be considered. Such events are caused by some internal or external events which have to be modelled appropriately. Also, hybrid approaches which can consider the both event sources types, exist.

B. Model time

Another difference between approaches is in how accurate the time is modelled.

In some approaches time can be only a source of cause-and-effect relationships between discrete events.

Duration of events and processes have to be taken into account for more accurate modelling. This allows to retrieve estimations of timing properties as a result of the model analysis.

C. How behaviour is modelled

Approaches of the discrete-event simulation can differ in the ways of how behaviour of model components is modelled.

Model can, for example, be represented as a *randomized events flow* with given probability characteristics and description of how the component reacts to external events. Model also can be imperative. For example, it can be represented as either a *finite state machine (FSM)*, extensions of FSM which work with extended memory state and time, or some other *transition systems*. Also the behaviour can be modelled as a *program model*, when model is a code in some programming language.

Finite state machines (in particular, extended and timed) and specialized transition systems are usually naturally suit for describing of a behaviour of small components. Also, some of such models are studied well and can be analyzed in some other way except the execution. Such analyses can be used during the whole system analysis. But still, usage of such models is a bad idea for modelling of complicated behaviours (in particular, requiring a lot of internal states and events).

Program models are vice versa: they can pretty conveniently used for modelling of very complicated behaviour but they usually can be used only for execution.

But program models have an additional advantage: any simpler model (e. g. FSM and other transition systems) can be translated automatically to a program model. This means that if a simulation system supports program models, simpler model types can be supported automatically.

Randomized events flow is sometimes a really good behavioural model type for some types of components. And in most cases this model representation also can be translated to a program model automatically.

D. Abstractness

Two independent metrics of abstractness of the system models can be distinguished.

One of them is a *structural abstractness*. Structurally abstract models have components which are going to be refined in the future development but at the moment their structure, number and properties of its subcomponents are not known.

Another factor is a *behavioural abstractness*: the way how accurate behaviour is modelled. This influences on how accurate different aspects are reflected:

- internal state of a component;
- influence of a component to the other ones;
- data which components are working with;
- time intervals between events.

Which model characteristics can be retrieved during the analysis process depend on the behaviour abstractness of a model.

Relative complexity of behavioural models are more or less the same in case when model is accurate by both factors and in case when model is abstract by both factors. If structurally abstract model is built behaviourally accurate, behavioural models of each model component can be very complex (both by internal state and by interaction with environment).

E. General requirements

Support of working with models represented in different abstraction levels is essential for the early model analysis and validation. In particular, it is really important to analyze structurally abstract and behaviourally accurate models. This allows to check single structure refinements and to find out incorrect ones.

To provide this, the way of how behaviour is modelled have to be convenient for complex behaviours. But still, simple behaviours in structurally accurate models have to be able to be defined in a simple way (which is convenient to be done using some formalisms like transition systems).

That's why we consider that in a context of early analysis and validation it is important for a simulation system to support program models. But still, support of translation of other representations is really important too.

The very important aspect of usability of such simulation systems is their main users — designing engineers and integrators — familiarity to program tools and languages or at least to used paradigms.

Moreover it is important to be able to conveniently represent an internal state of a component and to work with it in a behavioural model in a convenient way. Considering ideas from above the best candidate for that is some imperative high-level language which has libraries of collections, basic algorithms and other useful features.

Models of avionics systems can be very big. But nevertheless simulation of such systems have to be performed relatively fast.

Such models usually can be divided into some parts that rarely interact with each other. This makes to think that parallel simulation (with correct distribution of components to nodes) can be performed with an acceptable performance.

Nowadays it is important for a simulation system to be portable. This is good both for users and for the organization process of a parallel simulation. But this requirement can add some restrictions to the way how behaviour is modelled.

III. RELATED WORK

There are some work and open-source instruments related to the discrete-event simulation.

Some instruments are based on formalisms that require explicit declaration of all finite states of the modelled component and all transitions between those states (including the timing properties of transitions). For example, instruments adevs [1], PowerDEVs [2] and DEVsPy

[3] use pretty popular formalism DEVS (Discrete Event System Specification) [4]. Galatea [5] is based on a similar formalism. There are discrete-event simulation systems based on the Petri net, for example CPN Tools [6]. As it is said above, such models can be successfully used in accurate models but do not suit for describing complex behaviours in structurally abstract models.

There are some instruments that are using seldom used in avionics area functional languages (for example, Scala and Haskell used by Facsimile [7] and Aivika [8], [9]) or specific custom languages (for example, jEQN [10] for SimArch [11]).

Some instruments which are not in classes above (for example Tortuga [12], MASON [13], DESMO-J [14] and SimPy [15]) architecturally cannot be parallelized.

JaamSim instrument [16] is aimed only to graphical simulation and that's why it hardly can be used in the automated model analysis.

Thus, for fitting all requirements it is needed to design an architecture of discrete-event simulation which supports program models (using imperative high-level language) with parallel simulation support.

IV. ARCHITECTURE

A. Interaction with simulation environment

The first question was how to organize interaction between a behaviour model and its environment: simulation time and other model components.

We will call a *basis* a set of action types in a code of a program model which can be used to describe a behavior. A basis has a part intended for an interaction with environment.

Two fundamentally different approaches were considered.

1) *Synchronous interaction*: Originally a *synchronous basis* was chosen. This approach is used in a number of discrete-event simulation libraries.

The main idea of the approach is that the code of a behaviour model determines moments of time when it is ready in get information from outside. In that case behaviour model can contain the following types of actions (besides actions on the internal state):

- non-blocking data sending to some other model component;
- notification a simulator about the end of the current discrete event with the next event starting at:
 - given moment of the simulation time;
 - external event receiving (with ability to set a timeout).

Program models designed for this approach are linear. Such program model can be defined as a description of an internal state and a single function containing all behaviour. Such way of modelling seems to be natural

because a component life-cycle exists explicitly in a model as a single entity (function mentioned above).

Moreover, simulator providing such basis can be implemented to be very effective. In particular, such modelling approach allows to simulate independently rarely interacting parts of a model: until some parts do not send some data to each other they can be simulated in their own simulation time without violation of consistency and with no need of saving and storing of internal states.

But this approach has some problems.

Lets consider a situation when basis of interaction allows ending of a discrete event for receiving external data with a zero timeout. This is used, in particular, to implement a logical expression of kind "if at the current moment there was an external event X we should do one thing else another one". Such kind of expressions are widely used in behaviour modelling when using synchronous basis.

In the considered case results of a simulation really depends on the order of execution of discrete events which are scheduled to the same simulation time. In other words, some messages from one to another model component can be non-deterministically delayed. Obviously, simulation that allows such case cannot be used for accurate analysis of latency and other timing characteristics of models.

Basis can be changed in the following way to not allow this. Behaviour model have to set a *delta* — positive time which is the minimal time to wait before the next event can arise — each time the external events are processed. The problem of using of such basis can raise when such delta have to be null or depends on external data. In that case delta cannot be given correctly.

One more problem of this basis was discovered. The problem is that the code of behaviour model have to manage external data in the order of their coming. This is not always possible (or, at least, convenient) because sometimes component have to manage some concrete data to make a decision. This leads a basis to have methods for incoming data filtering.

2) *Asynchronous interaction*: An *asynchronous basis* is an alternative to a synchronous one. The behaviour model have to react somehow to an external data in the very moment it comes. But still, one of the variant of such reaction is ignoring.

It is important that a model represented in such basis is not linear. This leads a rethinking of what the component behaviour model is. Also it influences on an internal state of a component.

It is stated that any model represented in a synchronous basis can be represented in an asynchronous one. This means that the class of possible behaviours described in an asynchronous basis is not smaller (and, in fact, bigger) that class of synchronous ones.

Such basis allows to consider all incoming data and to interrogate other components each moment of time. This makes behaviour modelling much easier comparing to the synchronous approach.

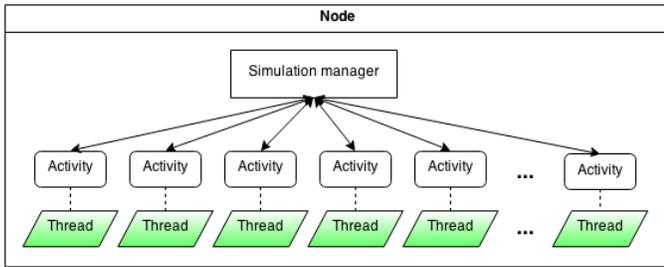


Fig. 1. Threaded approach on a single node

Ability of interrogation allows to model a continuous process initiation in an easy way. One component can initiate a continuous process in another component and is able to retrieve a result of this process as soon as it finishes.

Thus, asynchronous basis contains the following actions list:

- non-blocking data sending to some other model component;
- notification a simulator about the end of the current discrete event with beginning of the next event starting not later than the given moment of time;
- interrogation of the internal state of other component or initiation of a continuous process in it (in that case the current discrete event ending is also performed).

Simulator providing such basis is harder to implement in an effective way (comparing to the synchronous one). Nevertheless, the class of behaviours that can be modelled becomes adequate to requirements. That's why it was decided to use the asynchronous basis.

B. Execution architecture

It is important to consider that program model code execution have to be suspended at the end of discrete events to make other components to able to execute their own events at the same moment of the model time.

One of the simplest and obvious ways of organization of such alternating execution is the usage of a multiple threads. One thread is created for each component and they are suspended by a simulation system when a function of the end of discrete event is called. Thread is suspended until new event is raised for the corresponding component.

This approach is practical and pretty easy to implement. But it has some remarkable drawbacks.

One of them is that a behaviour model writer can easily create a deadlock. This situation can be preserved by using of some conventions for the behaviour model code but these conventions cannot be checked automatically by a simulation system.

But the main drawback, as it is seen from practise, is a high load to the threading subsystem of an operating system: models can have tens of thousands of active components so there are the same count of threads (fig. 1). This

worked well for Linux-based operating systems. But some operating systems cannot manage with such load which leads to inability of simulation using them. This approach made the portability to be a problem.

One way of how this problem can be solved is a parallel simulation. This means that if we have enough count of nodes each of them would have small enough count of threads to be managed by any multithreaded operating system (fig. 2).

But having the size of a model as tens of thousands of components and limitation of operating systems to about a hundred of threads we have to use hundreds of nodes. Sometimes it is not really possible to have such count of nodes. Nevertheless usually models cannot be divided to hundreds of parts which rarely interact. This means that overhead of such simulation will be very big.

However, another way of such models execution organization exists. This way does not have drawbacks mentioned above but still requires some effort to apply it. This approach is called *continuations* or *coroutines* in the computer science [17], [18].

The continuations approach allows to execute several different program models in a single system thread. This means that program model code can be suspended and after that it can be resumed from the very point it was suspended. In other words, the program model code can be run at the beginning of the current discrete event handling point.

This approach runs into a problem of correct error tracing because control flow changes vastly and some effort is needed to make error traces (including stack traces) to look as if control flow was unchanged. Storing of additional information for that leads having some overheads.

Some modern and progressive programming languages which are using virtual machines for program execution, have the continuations approach built in. Classic languages have libraries implementing this approach but these libraries require an after-compilation program instrumentation.

Instrumentation of library program models is not a hard problem. But instrumentation of user program models can be a problem and require additional organization of the simulation process start.

Nevertheless, applying this approach (fig. 3) allows to increase maximal count of model components running a single node. This count becomes operating system independent. This means that more optimal division of a model to simulation nodes can be achieved. Thereby simulation effectiveness increases.

V. INTEGRATION

Simulator having architecture described above has been implemented and integrated as a part of an instrument MASIW [19], [20]. This instrument is dedicated to developing and analysis of avionics models using AADL [21] (architecture analysis and design language), which is

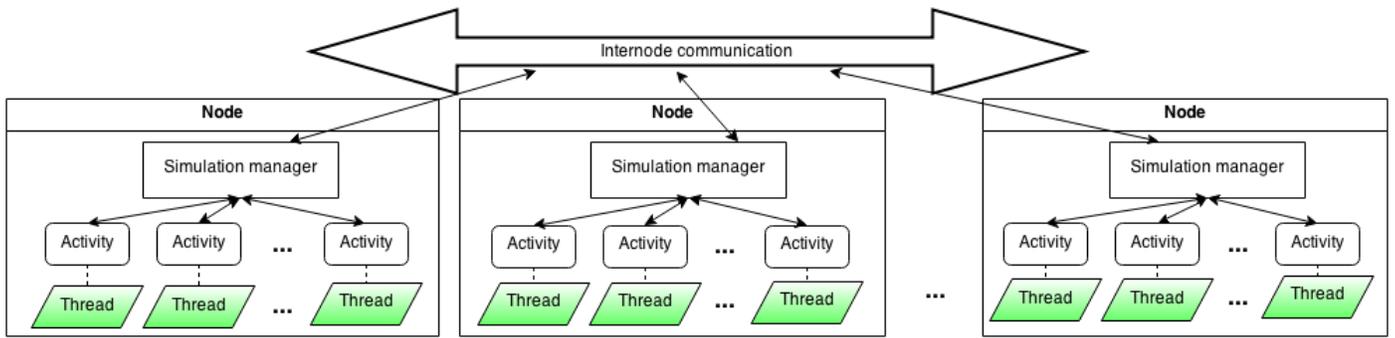


Fig. 2. Threaded approach on multiple nodes

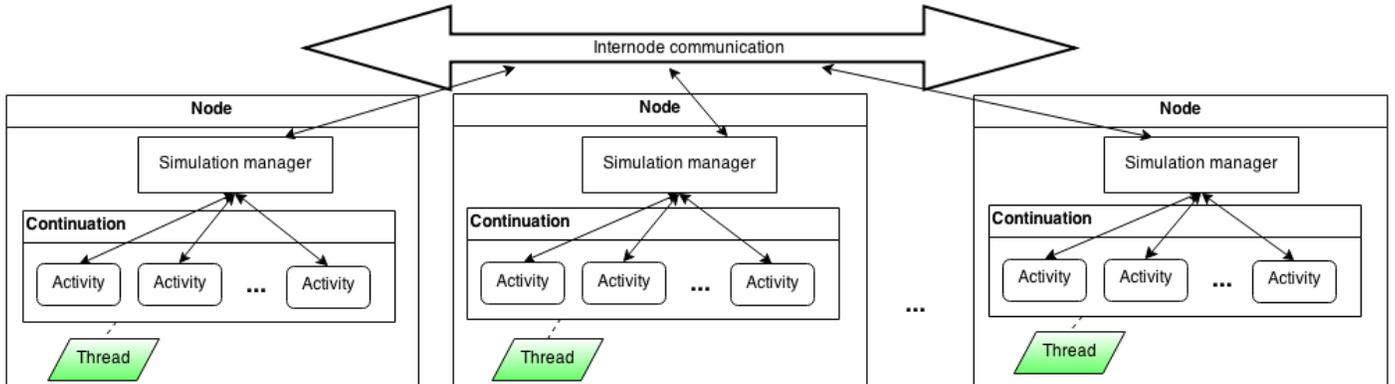


Fig. 3. Continuations approach on multiple nodes

widely used in creation of different responsible systems including avionics systems.

MASIW is a platform of developing of AADL-based models. It supports different representations and analysis of developed systems. The framework of this instrument is an open source.

This instrument has several static and dynamic model analysers. One of its analysers is a discrete-event simulator. It is based on an architecture above and is used for a general dynamic analysis of AADL-models' behaviour.

VI. CONCLUSION

The main contribution is the architecture of the discrete-event simulation system that allows to simulate large systems (containing tens of thousands of components) using program behaviour models.

Also interaction of a program model with a simulation environment was investigated. It was shown that some approaches of interaction used in simulation libraries are inapplicable in some cases. An alternative way was suggested.

These architecture solutions were applied in a powerful avionics model design and analysis tool. This allowed to perform pretty fast and accurate analysis of avionics models (including models for the early validation). The architecture allowed to not to require a lot of simulation nodes for such analysis.

REFERENCES

- [1] Adevs library, <http://web.ornl.gov/~lqn/adevs/>.
- [2] PowerDEVS, <http://sourceforge.net/projects/powerdevs/>.
- [3] DEVSImPy, <https://code.google.com/p/devsimpy/>.
- [4] B. Zeigler, *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. San Diego: Academic Press, 2000.
- [5] Galatea, <http://galatea.sourceforge.net/>.
- [6] CPN Tools, <http://cpntools.org/>.
- [7] Facsimile, <http://facsim.org/>.
- [8] D. Sorokin, *An Introduction to Aivika Simulation Library*, 2013, <https://github.com/dsorokin/aivika>.
- [9] A scala port of the Aivika simulation library, <https://github.com/dsorokin/scala-aivika>.
- [10] A. D'Ambrogio, D. Gianni, and G. Iazeolla, "jEQN a java-based language for the distributed simulation of queueing networks," in *Computer and Information Sciences — ISCIS 2006*, ser. Lecture Notes in Computer Science, A. Levi, E. Savaş, H. Yenigün, S. Balcısoy, and Y. Saygın, Eds. Springer Berlin Heidelberg, 2006, vol. 4263, pp. 854–865. [Online]. Available: http://dx.doi.org/10.1007/11902140_89
- [11] D. Gianni, A. D'Ambrogio, and G. Iazeolla, "SimArch: A layered architectural approach to reduce the development effort of distributed simulation systems," in *Proceedings of the 11th International Workshop on Simulation & EGSE Facilities for Space Programmes (SESP10)*, Noordwijk, The Netherlands, sep 2010.
- [12] Tortuga, <https://code.google.com/p/tortugas/>.
- [13] MASON Multiagent Simulation Tool, <http://cs.gmu.edu/~eclab/projects/mason/>.
- [14] DESMO-J, <http://desmoj.sourceforge.net/home.html>.
- [15] SimPy, <http://simpy.readthedocs.org/en/latest/>.

- [16] JaamSim, <http://jaamsim.com/>.
- [17] D. E. Knuth, *The Art of Computer Programming vol. 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley, 1997, pp. 193–200.
- [18] J. C. Reynolds, “The discoveries of continuations,” *Lisp and Symbolic Computation*, vol. 6, no. 3-4, pp. 233–248, 1993.
- [19] A. Khoroshilov, D. Albitskiy, I. Koverninskiy, M. Olshanskiy, A. Petrenko, and A. Ugnenko, “AADL-based toolset for IMA system design and integration,” in *SAE 2012 Aerospace Electronics and Avionics Systems Conference*, vol. 5, no. 2. SAE Int., 2012, pp. 294–299.
- [20] D. Buzdalov, S. Zelenov, E. Kornyxkin, A. Petrenko, A. Strakh, A. Ugnenko, and A. Khoroshilov, “Tools for system design of integrated modular avionics,” in *Proceedings of the Institute for System Programming of RAS*, vol. 26, no. 1, 2014, pp. 201–230.
- [21] *Architecture Analysis & Design Language (AADL)*, *SAE International standard AS5506B*, SAE International, 2012, <http://standards.sae.org/as5506b/>.

Protecting Applications from Highly Privileged Malware Using Bare-metal Hypervisor

Kurbanmagomed Mallachiev
Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russian Federation
mallachiev@ispras.ru

Nikolay Pakulin
Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russian Federation
npak@ispras.ru

Abstract—The paper presents a work-in-progress project on construction of a security facility that protects trusted application from malware residing at any privilege level of an OS, including OS kernel. The approach is based on the Sevigator project that used KVM to protect applications running in QEMU. The presented project is a port of Sevigator to much smaller trusted computing base of a bare-metal hypervisor.

Keywords—security, virtualization, confidentiality, hypervisor, protection, virtual machine monitor, Sevigator

I. INTRODUCTION

The purpose of the project is to develop a security facility, that protects data confidentiality on a computer connected to the Internet and managed by an untrusted operating system. We assume that malicious code can get unlimited access to all hardware and software system resources through vulnerability or backdoors in system software.

Modern widespread operating systems (such as Linux or Windows) are based on monolithic kernel, where all components of kernel have equal privileges. When malicious code penetrates OS kernel there is a risk of losing control over any OS resources including application in-memory data, confidential information in file storage, etc. Integrity and confidentiality of data transmitted over the network are also threatened, even in the case when cryptography is used.

There are several channels for malicious code to penetrate OS kernel. It could be vulnerability of the system applications and kernel vulnerabilities, and backdoors in the drivers. Also there is a risk of theft of private keys from companies, supplying software or hardware, to sign malicious code; as a result OS trusts the signature and installs such code in the kernel.

Multiple approaches to securing workstations were proposed, including new more secure operating systems, specific hardware extensions, new application architectures. Still those approaches require massive investments in new products and significant changes in the user experience.

The question is whether it is possible to protect unmodified applications that run under unmodified commodity OS like Windows or Linux on a commodity workstation with x86 CPU. Protection systems located in kernel, such as antivirus, firewall, intrusion detection, can themselves be attacked by

privileged malicious code. Possible way of protection from those attacks is the transfer of protection to more privileged level.

The answer is “probably yes”: a prototype called Sevigator [3, 4, 5, 6] protects applications in Linux from malware and comprised kernel. It uses hardware-assisted virtualization [1] to secure operating memory of applications and control access to communication hardware (network interface card). It allows to launch OS under control of virtual machine monitor (VMM, also called hypervisor). Hypervisor is much smaller than OS, fully isolated from it, and has higher privilege than OS. Hardware virtualization is supported by most modern processors, which suggests the possibility of widespread use of security systems based on hypervisors

One of the first examples of the use of virtualization to protect against untrusted OS is Overshadow project of memory protection developed by researchers in Stanford and Princeton Universities, MIT и VMware, Inc [2]. This technology does not require modification of the operating system or application. All memory of running processes is encrypted when a context switches. So, if the operating system or another program tries to read data from the memory of the process, they will receive only encrypted data, while the trusted process, referring to own data, receives it in the original form. However this approach limits cases when trusted application needs to pass some data to other processes by means, for example, of shared memory. Also in this approach all data are encrypted, even those that require no protection, and that is overkill.

Another reliable way to prevent data leakage, under the assumption that malicious code in the OS kernel, is the physical isolation of the computer from the network connection. However in this case all legitimate applications, which require access to the network, would suffer.

Sevigator isolates untrusted OS from network, but keeps operability of trusted application. For them, and only for them, an access to network resources is granted. An important feature of this approach is that there is no need to recompile any applications or OS

Within Sevigator approach OS resides in a virtual machine, while protection system is located in type 2 (hosted) hypervisor. It provides facilities to isolate untrusted applications from network access; to prevent data leaks due to

code intrusion or memory attacks it controls memory integrity of the applications under protection. Description of security algorithms can be found in [3, 4, 5, 6]. Sevigator system is based on hypervisor KVM (Kernel-based Virtual Machine)

Hypervisor KVM is type 2 hypervisor. Type 2(hosted) hypervisors runs like a module inside the host OS kernel, which handles interrupts, provides an abstraction of hardware and management of computer resources. Virtual machines with guest OS run like application in the host OS. Implementation based on this hypervisor is relatively simple, since such hypervisor allows you to develop and test on the same machine without rebooting, provides an opportunity to use debuggers and monitoring tools to find errors.

However, in this case, the hypervisor reliability depends on the operating system, which runs the hypervisor; in the KVM case it is Linux. The OS architecture is based on the principle of a monolithic kernel, so the hypervisor is vulnerable to attack by drivers and models of devices in the OS kernel. These defects do not exist in the decision based on type 1 hypervisor.

Type 1 hypervisors (native hypervisor, bare-metal hypervisor) run directly on the host's hardware. This hypervisor contain microkernel for interrupt processing, memory management, input-output, etc. Bare-metal hypervisors run at a higher privilege level than the OS kernel. A guest operating-system runs on the same privilege level as in the absence of the hypervisor.

Building protection systems based on type 1 hypervisor requires considerably less trusted computing base, than in the case of type 2 hypervisor. In addition, the microkernel allows you to split device drivers, virtual machines and memory manager. Thereby compromising individual component will not lead to compromise the entire system.

In this paper we present adaptation of part of Sevigator's protection algorithms, implemented in the type 2 hypervisor KVM, for type 1 hypervisor. Functionality of isolation OS and untrusted applications from network was adapted; currently being adapted security algorithms protecting process address space from unauthorized modification through the mechanism of direct memory access

II. CHOISE OF HYPERVISOR

When designing an adaptation to the type 1 hypervisor the idea to develop a hypervisor from scratch was immediately rejected: the development of a hypervisor is a very laborious task. It was necessary to compare existing type 1 hypervisors for x86 and select one to adapt functionality in it.

There are several requirements to hypervisors:

1. Open source. It is the base requirement to implement security mechanisms in the hypervisor code.
2. Support AMD x86 architecture, because, when we start adaptation, Sevigator used AMD virtualization
3. The presence of a virtual machine monitor to create and manage virtual machines and support of arbitrary unmodified guest operating systems.

4. Support for multiple virtual machines. Sevigator architecture assumes that at least two virtual machines run simultaneously.

5. Virtualization of hardware resources to separate the hardware between multiple virtual machines.

6. Small source code to allow verification.

The following hypervisors were considered: BitVisor[7], NOVA[8], Xen[9], XtratuM[10]. All of them are distributed under open source licenses and don't require existence of a host operating system.

BitVisor is hypervisor and virtual machine monitor, designed to ensure security of computer systems. BitVisor provides encryption of network connections and data on disk. Ensuring confidentiality of network and disk data is transparent to the operating system. BitVisor designed to create minimal overhead on encryption and decryption of data. BitVisor distributed under an open source license.

Virtual machine monitor is integrated into the hypervisor and performed at the same privilege level as the hypervisor. BitVisor supports exactly one virtual machine - this is done in order to minimize the overhead on the interaction of the guest OS with the devices, primarily input and output devices. BitVisor intercepts access to certain devices (eg, SATA controller, ie, hard disk), while the rest of the devices OS accesses directly.

BitVisor was rejected because it does not support multiple virtual machines.

NOVA is a hypervisor, built on microkernel architecture. Microkernel is performed at the highest level of privileges, and the environment, including resource monitor, device drivers and monitors virtual machines run at lower privilege levels. Thanks to microkernel architecture NOVA has well isolated code: components communicate with each other via messages, and with the kernel through hypercalls, only microkernel is performed with the highest privilege level, this provides improved security system as a whole.

Strictly speaking, the abbreviation NOVA used to refer to NOVA microkernel. In addition to the kernel running guest operating systems requires additional components developed in the project NUL (NOVA UserLand). NUL includes a virtual machine monitor Vancouver, memory and hardware resources monitor Sigma, external devices' drivers. Further in the text of this paper we will refer to NOVA bundled with NUL environment as just NOVA.

Originally the microkernel was developed at the Dresden University of Technology, now the main development of the kernel is in the research center of Intel.

NOVA is developed in C++, distributed under open source license. Using of microkernel architecture allows for simultaneous execution of an arbitrary number of virtual machines that can run unmodified guest operating systems. NOVA supports virtualization devices: Vancouver provides to guest OS virtual devices, which are served in the NUL. NOVA currently provides limited support for direct access to the

computer hardware, and limited support through separation devices IOMMU.

Xen is a very popular virtualization platform, which is widely used to build cloud services.

Xen virtualization platform includes a hypervisor, virtual machine monitor for guest OS, dedicated virtual machine dom0 to work with devices and specialized drivers to access the device via the dom0. These drivers are called paravirtualized as they "know" that the OS is running under Xen and effectively interact with the hypervisor and dom0.

Xen hypervisor implements the minimum set of operations: management of RAM, processor status, real time clock, interrupt processing and control of DMA (IOMMU). All other functions, such as the implementation of virtual devices, create and delete virtual machines, moving VMs between servers in the cloud, etc. is implemented in a dedicated virtual machine dom0.

All functions related to ensuring network performance, disk drives, video cards emulation and other devices placed outside the hypervisor. Typically, the request handling devices consist of two parts. Driver in the guest operating system translates requests from the OS to program handler in dom0. To increase the security of the system servers, virtualize devices run as separate processes in OS dom0. Failure in such a program leads to a denial of only one virtual device in one VM and does not affect the work of other copies of the server.

Xen hypervisor supports virtualization even on platforms where there is no hardware virtualization. As a result, the hypervisor code is quite large - on the order more microkernel NOVA - and convoluted. In addition, Xen does not support running unmodified guest OS: it requires specialized drivers to run the OS under Xen supervision.

XtratuM is hypervisor to separate computer resources into multiple virtual real time machines. XtratuM hypervisor provides real-time guarantees for the service interruptions hypercall, memory operations. XtratuM provides mechanisms isolation of virtual machines, the minimum software interface to run real-time applications without the guest OS, the means of communication between VMs. Developers claim support of x86 architecture, but the official website of the project distributes documentation for LEON processor family only – specialized clones if SPARC architecture.

XtratuM hypervisor is supported by several real time operating systems. Guest OS requires paravirtualized drivers, XtratuM does not support execution of arbitrary unmodified guest OS.

On the basis of requirements to the hypervisor we selected NOVA as the platform for bare metal hypervisor with security functions ported from Sevigator hypervisor.

III. SEVIGATOR ARCHITECTURE

Among the applications running in the operating system, the protection system identifies several applications that are considered trusted. The specific mode of functioning is provided to these applications. All other applications are considered as untrusted, the security problem is to prevent the

leakage or compromising of confidential data of trusted applications. In particular, trusted applications for the normal functioning may require access to the public network. This network connection in the absence of external control can be used by malicious code in the kernel of the operating system for the leakage of sensitive data. The task of the security is to prevent data leakage.

The solution is based on use of hardware virtualization technology, execution of an operating system and all software in the virtual machine, and implementation protection system in the body of a virtual machine monitor (hypervisor) [3]. The hypervisor provides simultaneous execution of two completely isolated from each other virtual machines (fig. 1). Both are running the same untrusted operating system. The first virtual machine, we will call it *private*, is the primary one. It is there where critical data resides, applications are executed (both trusted and untrusted), processing those data. When the private virtual machine starts hypervisor blocks access to the network interface. The operating system, which runs in the VM, believes that the network adapter is physically absent. Therefore, any attempt to establish a network connection from within the virtual machine and transmit the data to a remote computer will inevitably lead to error. Thus, the malicious code running on any hardware privileges inside the private virtual machine, even if it managed to gain access to critical data, will not be able to transfer them to the outer world.

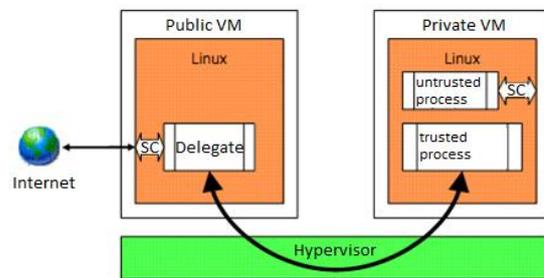


Fig. 1. Sevigator architecture

Network access for trusted applications is supplied by the second virtual machine. From here on we will refer to it as *public*. Public virtual machine has free access to the network interface, and any program in this virtual machine can interact with remote computers on the network. However, due to virtual machines isolation provided by the hypervisor the software in the public virtual machine (including Linux kernel) cannot gain access to data residing within the private virtual machine.

Network support for trusted processes is implemented through remote execution of required (limited) set of system calls to the public virtual machine. The hypervisor intercepts system calls invoked by a trusted process, analyzes the data and, when necessary, transmits them to the public virtual. System calls of other processes as well as the rest of the system calls of trusted processes are serviced locally in the private virtual machine.

The only information transmitted outside the private virtual machine is explicitly specified by a trusted process as parameters of system calls, and besides transfer the information outside of the virtual machine is serviced by trusted code

(hypervisor). Note that the remote execution of the system call is made transparent for a trusted process and an operating system for a virtual computing machine.

Trusted processes are executing under the control of an untrusted operating system. In-memory data of trusted applications are not encrypted, stored in clear (unencrypted) form, and protection system does not restrict the access (both system and user) to these data. Since untrusted components, including the kernel, do not have access to the network, they are not able to disclose sensitive information.

However, in untrusted OS environment it is necessary to take into account the risk of injecting code into trusted applications: malicious code in the operating system kernel can load into the address space of a trusted process necessary code, then pass control to it, and trusted process on its behalf will take all necessary actions for the delivery critical information to a remote computer, which is controlled by the attacker. To prevent these harmful effects security system protects context of a trusted process against unauthorized modification by any program in private virtual machine, including privileged.

IV. ADDAPTATION FROM KVM TO NOVA

So, as mentioned above, open source NOVA hypervisor has been selected as the hypervisor. At the moment it is being actively developed in the Intel Research laboratory.

By the arguments in favor of NOVA, above, may be added that the hypervisor much less than all popular hypervisors therefore potentially more secure. Also its kernel code has been verified [11]. On the Fig. 2 you can see a comparison of the sizes of popular hypervisors and NOVA.

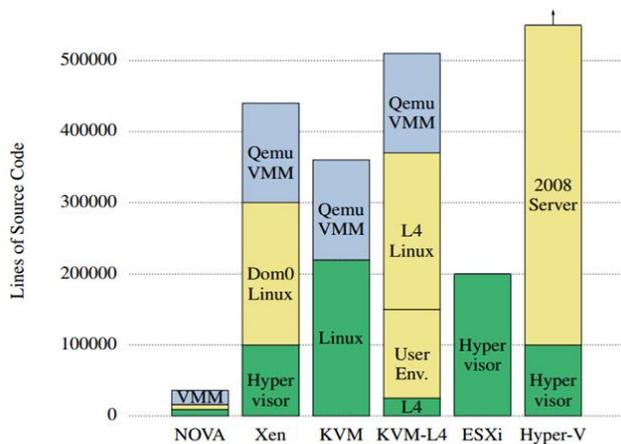


Fig. 2. comparison of the sizes of popular hypervisors

NOVA is built on microkernel architecture. Microkernel, which size is less than 10,000 lines of C ++, launches virtual machines and routes interrupts and system calls. In addition to the core NOVA includes the “Nova UserLand” NUL, which includes a virtual machine monitor (Vancouver), memory and hardware resources monitor (Sigma0) and drivers of external devices. The total size of NOVA and NUL is less than 50 thousand lines.

Sevigator was buit on top of KVM kernel module that provides hardware virtualization in QEMU environment. Intercepting calls of the virtual machines, it entrusts to processing of many functions host OS kernel, under which it is launched. Moreover KVM is included as a module in the host OS kernel, and therefore has the highest level of privilege. There is no division of privilege levels in KVM. The size of KVM is over 300 thousand lines of code. In the prototype Sevigator based on KVM interaction was through the address space and virtual interrupt of pci device, which emulated by qemu.

During transferring Sevigator to NOVA platform most of Sevigator’s algorithms have been implemented in components of NUL.

This section briefly describes the changes made to the NOVA and NUL, to implement algorithms from Sevigator.

After transfer Sevigator to NOVA a remote service system call will appear as diagram in Fig. 3

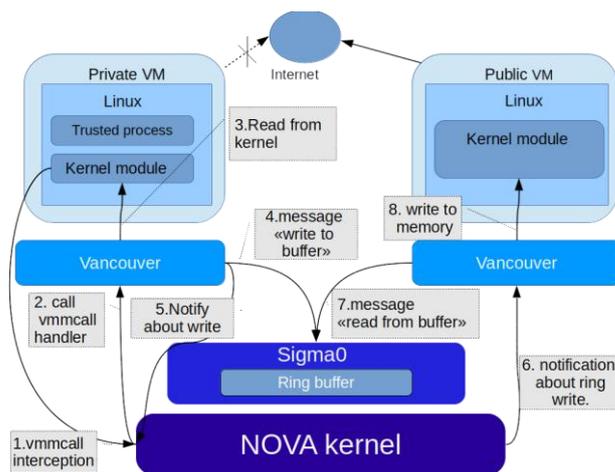


Fig. 3. Architecture of Sevigator in NOVA.

At system start the virtual machine monitor, controlling user (private) VM, is configured so that it does not have a network card emulator. That is, when the OS at boot enumerates PCI bus, it lacks the class of device "network device". So, operating system has no access to the network, and all untrusted components, including the kernel, cannot transmit and receive data from the outside.

However, when a trusted application performs a system call related to network access, the system call is intercepted by the hypervisor. System call parameters are copied into an internal ring buffer, which is not accessible by guest OS in virtual machines. VM monitor service receives notification of a new data, reads data from the ring buffer and transmits to the public VM for handling. Transferring the data from the public VM to the private VM is implemented similarly.

System calls are intercepted by the hypervisor, but the processing parameters of the call, the data transmission between the VM, return control to the VM implemented in virtual machine monitor. This is done to improve the security of the system: if the query processing network has vulnerability then only one VM has been compromised, all other processes

are executed as a microkernel's process and isolated from each other.

Functions that implement algorithms of Sevigator were added to the virtual machine manager Vancouver, which runs as an application process in NOVA. This protects hypervisor itself from compromising by the security system. Even if malicious code can take control over Sevigator, in a single VM, it will not be able to subdue other virtual machines.

V. MODIFICATION OF NOVA AND NUL

A number of changes to the core components of NOVA and NUL was required to properly implement the Sevigator.

1. Sevigator's components, working in the OS kernel and user space (trusted applications launcher), interact with the hypervisor through the vmmcall instruction. It was necessary to implement handlers of vmmcall in NUL. For this, interception vmmcall instruction has been activated in the NOVA's kernel and implemented handler for instruction in virtual machines manager Vancouver. On AMD platform instructions interception implemented by setting to 1 the bit, responsible for this instruction in a control block of the virtual machine (VMCB). The analysis showed that by the default NOVA set to 1 bits responsible for intercept vmload, vmsave, clgi, skinit. In order not to break code integrity the special flag has been added and the specific mask has been changed.

2. To trace action of the operating system, primarily context switching, Sevigator intercepts specific x86 instructions. These are system calls, software interrupts, returns from the interrupt, and others. To support it special handlers, embedded in KVM code, are implemented in the prototype. When moved to NOVA it is necessary to implement these instructions interception means of Sevigator. While intercepting instruction, you must be able to emulate, as the interception occurs before running the instruction [12]. Currently Vancouver does not have complete emulation necessary instructions (iret, int n, syscall, sysexit), but the implementation of the emulation is planned by the developers. Since emulation instructions was not included in the plan of this work, we decided to temporarily inserted into the core of the guest virtual machine vmmcall calls immediately after a system call and before returning from the system call. This decision violates the initial concept of protection without making changes to the OS, but it is temporary and it will be replaced as soon as Vancouver developers implement a full emulation of instructions.

3. To implement memory protection, we need to be able to access to the virtual machine memory. In Vancouver there is a subsystem, responsible for emulation instructions that work with memory. It has no external interfaces through which the other subsystem (in particular Sevigator) can work with memory. To access the virtual machine's memory by virtual address have been added a special message for reading and writing to the memory of virtual machines. Recall that the interaction between the components of NUL is given by messaging. Accordingly, new types of message for reading and writing were added, as well as handlers of these messages to the subsystem operating with the memory.

4. To implement the algorithms of the Sevigator's network subsystem should be developed mechanism and implemented

software interface of communication subsystem between virtual machines NUL. We implemented a ring buffer between managers of virtual machines, and notification of arrival a new data signaled through interruption to the virtual machine. In order to do this, special messages (OP_SVG_WRITE, OP_SVG_READ) handlers were added to operation memory monitor. And for notification, during initialization both Vancouvers initialize its own portals, and send its addresses to the NOVA kernel; NOVA has handler of the specific system call, which sends a signal to the desired portal. After storing data to ring buffer Vancouver needs to send a special system call to the kernel NOVA, which will notify the other Vancouver about recording.

5. Sevigator assumes access to the network card. In NUL possibility of direct access to the pci-devices are implemented only partially. NOVA has a feature to be the network provider for the virtual machine, but to use it you need to write a driver for the network card. We based our driver on the driver, which was included to the NUL to work with a network card family ne2000 that emulated by qemu. This driver has been slightly modified to work with a specific physical card of the ne2000 family. The changes are minimal – we added a missing flag and reduced the size of the buffer.

VI. CONCLUSION

In this paper we presented an implementation of privacy protection of network connection of trusted Linux processes in bare-metal hypervisor. We have conducted a study of open source bare-metal hypervisors: we have formulated certain requirements to the type 1 hypervisor, analyzed the available open hypervisors and by comparing have selected hypervisor NOVA.

Porting Sevigator from KVM to NOVA is in progress, we have partly ported confidentiality protecting mechanisms. We have conducted a study of architecture and mechanisms of components interaction within NOVA. We have identified and implemented changes in NOVA, necessary for the functioning of Sevigator. This allowed:

- to port mechanisms of remote service of system calls;
- to port mechanisms to control trusted application context;
- to port mechanisms to control memory integrity of trusted applications;
- implement a prototype of network boot mechanism.

Thus, the outcome of the project is a prototype of security system that protects the integrity and confidentiality of information stored and processed on a computer connected to the network and controlled by potentially malicious operating system.

Developed prototype provides protection of information systems from the injection of malicious code by modifying the memory, images of executable files or script files, protect the integrity and confidentiality of data while transiting over the network.

- [1] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide [Online]. Available: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf>
- [2] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 2–13, March 2008. [Online]. Available: <http://doi.acm.org/10.1145/1353535.1346284>
- [3] Burdonov I., Kosachev A., Iakovenko P. Virtualization-based separation of privilege: working with sensitive data in untrusted environment. //1st Eurosys Workshop on Virtualization Technology for Dependable Systems, New York, NY, USA, ACM. 2009. P. 1-6.
- [4] D.V. Silakov. Using Hardware-assisted Virtualization in the Information Security Area. pp. 25-36. Proceedings of the Institute for System Programming of RAS, volume 20, 2011. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)
- [5] P. Iakovenko. Transparent mechanism for remote system call execution. pp. 221-242. Proceedings of the Institute for System Programming of RAS, volume 18, 2010. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print)
- [6] P. Iakovenko. Ensuring confidentiality of information processed on a computer with a network connection. Information security problems. Computer Systems. №4. 2009. pp. 23-41. (In russian)
- [7] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. 2009. BitVisor: a thin hypervisor for enforcing i/o device security. In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '09). ACM, New York, NY, USA, 121-130.
- [8] Udo Steinberg and Bernhard Kauer. 2010. NOVA: a microhypervisor-based secure virtualization architecture. In Proceedings of the 5th European conference on Computer systems (EuroSys '10). ACM, New York, NY, USA, 209-222.
- [9] Chris Takemura and Luke S. Crawford. The Book of Xen. No Starch Press. October 2009, 312 pp. ISBN-13 978-1-59327-186-2,
- [10] A. Crespo, I. Ripoll, and M. Masmano. 2010. Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In Proceedings of the 2010 European Dependable Computing Conference (EDCC '10). IEEE Computer Society, Washington, DC, USA
- [11] Nova Micro-Hypervisor Verification http://os.inf.tu-dresden.de/papers_ps/tr-tews-vnova-2008.pdf
- [12] AMD64 Architecture Programmer's Manual Volume 2: System Programming http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf

Checking Conformance of High-Level Business Process Models to Event Logs

Antonina K. Begicheva
National Research University
Higher School of Economics (HSE)
33 Kirpichnaya Str., Moscow, Russia
Email: be-ton@yandex.ru

Irina A. Lomazova
National Research University
Higher School of Economics (HSE)
33 Kirpichnaya Str., Moscow, Russia
Email: ilomazova@hse.ru

I. INTRODUCTION

Process mining [1] is a new technology, providing a variety of methods to discover, monitor and improve real processes by extracting knowledge from event logs. The two most prominent process mining tasks are: (i) process discovery: constructing a process model from example behavior recorded in an event log, and (ii) conformance checking: diagnosing and quantifying discrepancies between observed behavior and modeled behavior. There are many software products which allow us to use methods of Process Mining. ProM [4] is an open-source tool supporting many techniques of Process Mining, which are represented as plug-ins. Due to a flexibility of this environment it can be used both for reserch and applications.

This paper studies conformance checking [1], [3], [6], [7]. Conformance checking uses both an event log and a model, and compares observed behavior written in the log with the behavior produced by the model. The general goal is to find discrepancies between them to improve a model. Conformance checking techniques can also be used for measuring the performance of process discovery algorithms (that restores a model on the basis of a known log) and to repair models that have no a well alignment with the real behavior of the process.

There are four model's evaluation criteria: fitness, precision, generalization and simplicity. Fitness measures "the proportion of behavior in the event log possible according to the model". Among the four quality criteria, fitness is the most related to the conformance. Several methods of conformance checking were developed. We consider methods based on *replay* approach [3]. Replaying a log on a model can help to measure *fitness*.

An obvious approach to measure fitness would be just to count the fraction of cases that can be "parsed completely" (i.e. the proportion of cases corresponding to firing sequences leading from *[start]* to *[end]*). Fitness can range from 0 to 1. It is supposed that fitness is equal to 1, if the log *perfectly fits* the model. When measuring fitness by replaying, we could stop replaying a trace when we face a problem and mark this trace as unsuitable. We get more information about conformance if we continue replaying the trace on the model, and record a count of all missing tokens and all tokens that are pending at the end.

Let us denote the number of produced tokens by p , the number of consumed tokens by c , the number of missing tokens by m and the number of remaining tokens by r . Initially,

when all places are empty, $p = c = 0$. Then the environment produces a token for the place *[start]*. Therefore, the p counter is incremented: $p \leftarrow p + 1$. A log is replayed by consecutive firings of transitions, corresponding to activities of the process. Each transition consumes and produces several tokens and we increase the corresponding variables. If we need an extra token in a place to continue replaying (when the next transition is not enabled), then the m counter must be incremented and the place, that lacks a token, is marked as a place where a token was missed. If by the time of consuming a token from the place *[end]* there were tokens pending in some other places, the r counter must be increased by the number of the remained tokens, and the places with tokens must be tagged.

The fitness of a trace σ of a workflow process model N is defined as follows:

$$fitness(\sigma, N) = \frac{1}{2}(1 - \frac{m}{c}) + \frac{1}{2}(1 - \frac{r}{p})$$

When working with business processes we typically use detailed logs, which present the full report about sequentially executed activities. Since in most information systems logs are generated automatically, keeping detailed records is not a problem. However, large and detailed models are not good to deal with. Such models are not clear and readable for experts. Experts prefer to work with more abstract (high-level) models. More abstract models are easier to construct, understand and analyze. Process models developed by people are, as a rule, not very large and abstract from technical details. So, checking conformance of an abstract model and a low-level event log, generated by an information system, is an important and challenging problem. However, as far as we know, this problem was not studied in the literature.

In this paper we consider an abstract model in which each separate activity represents a subprocess built from a set of smaller activities. A history of a detailed process behavior is recorded in low-level logs. Process models are represented by workflow nets — a special subclass of Petri nets [2]. We present a method for checking conformance of an abstract model and a low-level event log.

The paper is organized as follows. Section II contains some basic definitions and notions, including Petri nets, event log, perfect fits and refinement. In Section III we give a motivating example of handling a request for a compensation within airline in terms of Petri nets. In Section IV we present a method for checking conformance between an abstract model and a low-level log. We also give a justification of this method

by proving its correctness in the case of perfect fitness. An implementation of our algorithm is described in Section V. Section VI contains some conclusions.

II. PRELIMINARIES

We start with recalling some basic notions from the set theory. Let S be a set. By S^* we denote the set of all finite sequences (words) over S .

$S = S_1 \cup S_2 \cup \dots \cup S_n$ is a *partition* of S iff $\forall i, j \in [1, n] : S_i \subseteq S$ and $S_i \cap S_j = \emptyset$.

A *multiset* m over a set S is a mapping $m : S \rightarrow \text{Nat}$, where Nat is the set of natural numbers (including zero), i.e. a multiset may contain several copies of the same element.

For two multisets m, m' we write $m \subseteq m'$ iff $\forall s \in S : m(s) \leq m'(s)$ (the inclusion relation). The sum of two multisets m and m' is defined as usual: $\forall s \in S : (m + m')(s) = m(s) + m'(s)$, the difference is a partial function: $\forall s \in S$ such that $m(s) \geq m'(s) : (m - m')(s) = m(s) - m'(s)$. By $\mathcal{M}(S)$ we denote the set of all finite multisets over S . Non-negative integer vectors are often used to encode finite multisets.

Definition 1 (Petri net). Let P and T be disjoint finite sets of *places* and *transitions* and $F : (P \times T) \cup (T \times P) \rightarrow \text{Nat}$. Then $N = (P, T, F)$ is a *Petri net*. Let A be a finite set of activities. A labeled Petri net is a Petri net with a labeling function $\lambda : T \rightarrow A \cup \{\epsilon\}$ which maps every transition to an activity (a transition label) from A , or a special label ϵ , corresponding to an invisible action.

A *marking* in a Petri net is a function $m : P \rightarrow \text{Nat}$, mapping each place to some natural number (possibly zero). Thus a marking may be considered as a multiset over the set of places. Pictorially, P -elements are represented by circles, T -elements by boxes, and the flow relation F by directed arcs. Places may carry tokens represented by filled circles. A current marking m is designated by putting $m(p)$ tokens into each place $p \in P$.

For a transition $t \in T$ an arc (x, t) is called an *input arc*, and an arc (t, x) — an *output arc*; the *preset* $\bullet t$ and the *postset* $t \bullet$ are defined as the multisets over P such that $\bullet t(p) = F(p, t)$ and $t \bullet(p) = F(t, p)$ for each $p \in P$.

A transition $t \in T$ is *enabled* in a marking m iff $\forall p \in P m(p) \geq F(p, t)$. An enabled transition t may *fire* yielding a new marking $m' =_{\text{def}} m - \bullet t + t \bullet$, i. e. $m'(p) = m(p) - F(p, t) + F(t, p)$ for each $p \in P$ (denoted $m \xrightarrow{t} m'$, $m \xrightarrow{\lambda(t)} m'$, or just $m \rightarrow m'$).

A *Workflow-net* is a (labeled) Petri net with two special places: i and f . These places are used to mark the beginning and the ending of a workflow process.

Definition 2 (Workflow net). A (labeled) Petri net $N = (P, T, F, \lambda)$ is called a *workflow net (WF-net)* iff

- 1) There is one source place $i \in P$ and one sink place $f \in P$ s. t. $\bullet i = f \bullet = \emptyset$;
- 2) Every node from $P \cup T$ is on a path from i to f .
- 3) The initial marking in N contains the only token in its source place.

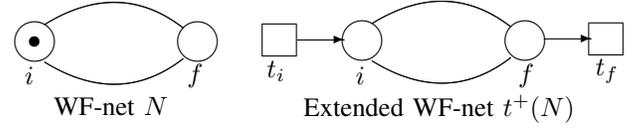


Fig. 1. Extending a WF net with initial and final transitions

By abuse of notation we denote by i both the source place and the initial marking in a WF-net. Similarly, we use f to denote the final marking in a WF-net N , defined as a marking containing the only token in the sink place f .

Let $N = (P, T, F, \lambda)$ be a WF-net. The *extended WF net (EWF-net)* $N' = (P', T', F', \lambda')$ is defined as follows: $P' = P, T' = T \cup \{t_i, t_f\}$, and $F' = F \cup \{\langle t_i, i \rangle, \langle f, t_f \rangle\}$, where t_i, t_f are new (not occurring in P, T) nodes. The new transitions t_i, t_f are labeled with invisible activity ϵ in N' , all other transitions in N' have the same labels as in N . In the remainder we will denote such an extended WF net of N as $t^+(N)$. The initial marking in an extended WF net contains no tokens. Thus an extended WF net may start a new case at any moment (cf. Fig.1).

Event logs keep a history of process executions.

Definition 3 (Event log). Let A be a finite set of activities. A *trace* σ is a finite sequence of activities, i.e., $\sigma \in A^*$. An *event log* L is a finite multiset of traces, i.e., $L \in \mathcal{M}(A^*)$.

In this paper we study conformance checking. Given a model and an event log we would like to compare the process model behavior and the behavior recorded in the event log. Several metrics for conformance checking were defined in the literature [1]. Among the most important metrics is *fitness*. Informally speaking, fitness measures the proportion of behavior in the event log possible according to the model.

Definition 4 (Perfect fit). Let N be a WF-net with transition labels from A , an initial marking i , and a final marking f . Let σ be a trace over A . We say that a trace $\sigma = a_1, \dots, a_k$ *perfectly fits* N iff there exists a sequence of firings $i = m_0 \xrightarrow{t_1} \dots \xrightarrow{t_k} m_{k+1} = f$ in N , s.t. the sequence of activities $\lambda(t_1), \lambda(t_2), \dots, \lambda(t_k)$ after deleting all invisible activities ϵ coincides with σ . A log L *perfectly fits* N iff every trace from L perfectly fits N .

Petri nets can be extended with hierarchy and it is done e.g. in Colored Petri nets (CPN) [8]. Hierarchy allows to develop more compact models with a compositional network structure. In the case of two-level hierarchy there are two models of one process: a high-level (*abstract*) model and a low-level (*refined*) model. The high-level model is a model with abstract transitions. An abstract transition refers to a Petri net subprocess model refining the activity represented by this transition. The low-level model can be obtained from an abstract model by substituting subprocess models for abstract transitions.

Definition 5 (Substitution). Let $N_1 = (P_1, T_1, F_1, \lambda_1)$ be a WF-net, $t \in T$ be a transition in N_1 . Let also $N_2 =$

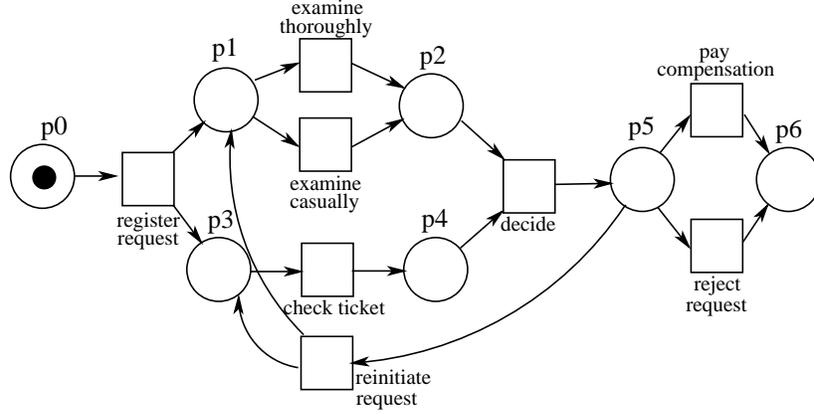


Fig. 2. An abstract model for handling compensation requests

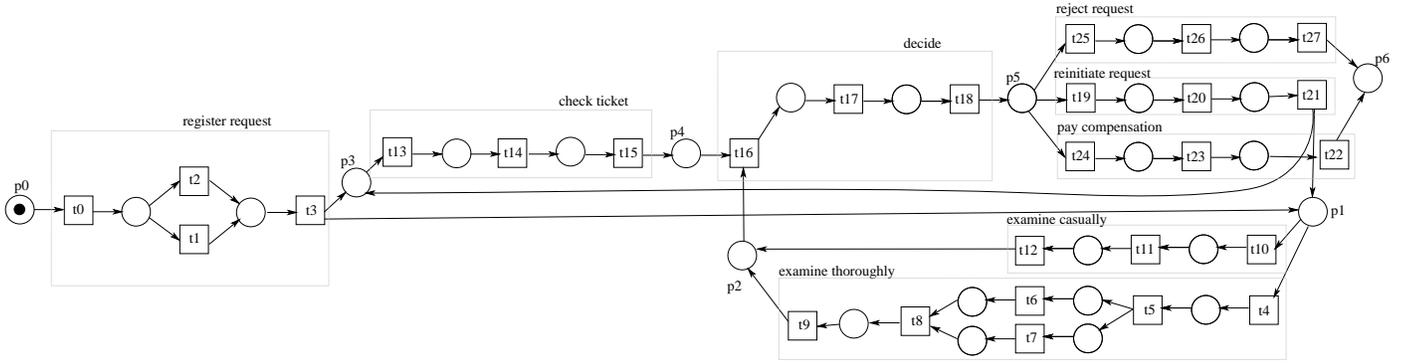


Fig. 3. A refined model for handling compensation requests, which refines the model in Fig. 2

$(P_2, T_2, F_2, \lambda_2)$ be an EWF-net with the initial and final transitions t_i, t_f correspondingly. We say that a WF-net $N_3 = (P_3, T_3, F_3, \lambda)$ is obtained by a *substitution* $[t \rightarrow N_2]$ of N_2 for t in N_1 iff $P_3 = P_1 \cup P_2$, $T_3 = T_1 \cup T_2 \setminus \{t\}$, $F_3 = F_1 \cup F_2 \setminus \{(p, t) \mid p \in \bullet t\} \setminus \{(t, p) \mid p \in t^\bullet\} \cup \{(p, t_i) \mid p \in \bullet t\} \cup \{(t_f, p) \mid p \in t^\bullet\}$,

Definition 6 (Refinement). Let N_a, N_r be two WF-nets with sets of activities A_a, A_r correspondingly. Let $A_a = a_1, a_2, \dots, a_n$, and $A_r = A_r^1 \cup A_r^2 \cup \dots \cup A_r^n$ be a partition of A_r into n subsets, and N^1, N^2, \dots, N^n be EWF-nets with sets of activities A^1, \dots, A^n correspondingly. We say that N_r is a refinement of N_a via substitutions $[a_1 \rightarrow N^1, a_2 \rightarrow N^2, \dots, a_n \rightarrow N^n]$ iff N_r can be obtained from N_a by simultaneous substitutions of N_r^i for all t s.t. $\lambda(t) = a_i$.

III. MOTIVATING EXAMPLE

Let us consider a toy model from [1], which describes handling a request for a compensation within airline. Here customers may request compensations for various reasons. An abstract model of this process (expressed in terms of a Petri net) is presented in Fig.2. Fig.3 presents a refined model of the same process. To avoid congestion of activities' names in the low-level model in Fig.3 only places inherited from the abstract model are labeled in the picture.

Let us explain the correspondence between the two models. Let $N^0, N^1, N^2, N^3, N^4, N^5, N^6, N^7$ be EWF-

nets with sets of activities $A^0 = \{t0, t1, t2, t3\}, A^1 = \{t4, t5, t6, t7, t8, t9\}, A^2 = \{t10, t11, t12\}, A^3 = \{t13, t14, t15\}, A^4 = \{t16, t17, t18\}, A^5 = \{t19, t20, t21\}, A^6 = \{t22, t23, t24\}, A^7 = \{t25, t26, t27\}$ correspondingly. The refined model in Fig. 3 is the refinement of the abstract model in Fig. 2 via the substitutions $[register_request \rightarrow N^1, examine_thoroughly \rightarrow N^2, examine_casually \rightarrow N^3, check_ticket \rightarrow N^4, decide \rightarrow N^5, reinitiate_request \rightarrow N^6, pay_compensation \rightarrow N^7, reject_request \rightarrow N^7]$. A sample of an event log obtained for the refined model L_r is shown in Fig.4. Note that for the log L_r (Fig.4) and the refined model (Fig.3) we have $fitness = 1$, since the log L_r is generated by the model.

So, we have an abstract model (as a more simple to understand and analyse) and we want to check conformance of this model to a low-level log. It is obvious that we cannot do it straightforward, since the model is defined in terms of abstract activities, and the log contains low-level activities.

IV. CHECKING CONFORMANCE BETWEEN AN ABSTRACT MODEL AND A REFINED EVENT LOG

To check conformance between an abstract model and a low-level log, we first transform the given log into a log over abstract activities. For this purpose, each low-level activity in the log is replaced by a name of the subprocess (an abstract

$L = \{ \langle t0, t1, t3, t4, t5, t6, t13, t14, t7, t8, t15, t9, t16, t17, t18, t25, t26, t27 \rangle,$
 $\langle t0, t1, t3, t4, t13, t14, t5, t7, t15, t6, t8, t9, t16, t17, t18, t24, t23, t22 \rangle,$
 $\langle t0, t1, t3, t13, t10, t11, t14, t15, t12, t16, t17, t18, t24, t23, t22 \rangle,$
 $\langle t0, t2, t3, t13, t4, t14, t15, t5, t6, t7, t8, t9, t16, t17, t18, t25, t26, t27 \rangle,$
 $\langle t0, t2, t3, t4, t13, t14, t15, t5, t7, t6, t8, t9, t16, t17, t18, t24, t23, t22 \rangle,$
 $\langle t0, t1, t3, t10, t11, t13, t12, t14, t15, t16, t17, t18, t19, t20, t21, t10, t11, t13, t14, t15, t12, t16, t17, t18, t25, t26, t27 \rangle,$
 $\langle t0, t2, t3, t13, t10, t14, t15, t11, t12, t16, t17, t18, t24, t23, t22 \rangle,$
 $\langle t0, t2, t3, t13, t10, t11, t12, t14, t15, t16, t17, t18, t19, t20, t21, t10, t13, t14, t11, t15, t12, t16, t17, t18, t24, t23, t22 \rangle,$
 $\langle t0, t2, t3, t13, t14, t10, t11, t15, t12, t16, t17, t18, t25, t26, t27 \rangle \}.$

Fig. 4. An event log for the refined model in Fig. 3

activity) it belongs to. Hence we get a log with "stuttering" abstract activities. This transformation is implemented by the method *toHighLevel()*, schematically presented in Algorithm 1.

Data: *lowlevellog* — a list of low-level activities,
hlaction — a set of high-level activities, where for each high-level activity is stored information about its partition into subsets of low-level activities.

Result: *highlevellog* — a high-level event log.

$i \leftarrow 0;$

$highlevellog \leftarrow \emptyset;$

$currentLowAction \leftarrow lowlevellog[i]$ **while** $i < lowlevellog.size$ **do**

```

// search of high-level activity,
// subsets of which contains this
// low-level activity
currentHighAction ←
search(hlaction, currentLowAction);
if currentHighAction ≠ ∅ then
    // check of condition that
    // low-level activity is included
    // to partition of the current
    // high-level action
    while  $i < lowlevellog.size$  and
    currentHighAction.contains(currentLowAction)
    do
        |  $i \leftarrow i + 1;$ 
    end
    highlevellog.add(currentHighAction);

```

end

end

return *highlevellog*;

Algorithm 1: Method *toHighLevel()*, transforming a low-level log into a log over abstract activities

After converting the refined log into notations of the abstract model we get a new log, which is a multiset of sequences of abstract activities. But this still can not be used for the conformance checking because of stuttering actions. Moreover, when we have two concurrent subprocesses, represented by two concurrent abstract activities in an abstract model, stuttering sequences may interleave. To overcome this problem we transform an abstract model into a model allowing stuttering of each abstract activity. For this purpose we add

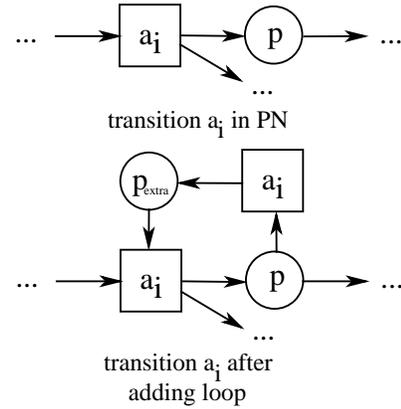


Fig. 5. Extending a transition by adding loop.

loops to transitions in the abstract model.

Algorithm 2 schematically describes the method *addLoops()* for transforming an abstract model by adding loops to abstract transitions (cf. Fig.5).

Now we describe the general algorithm of conformance checking between an abstract model and a low-level log in more details.

Main Algorithm (Converting a refined log to an abstract one and transforming the high-level model for working with result of this conversion).

Let $N_a = (P, T, F, \lambda)$ be a Petri-net corresponding to an abstract model of a process over a set of activities A_a . Let also L_r be a low-level event log (a finite multiset of traces) over a set A_r of low-level activities.

Let A_a, A_r be the set of activities in the abstract model and a set of activities in a refined model correspondingly. Denote by σ and denote trace from it by $\sigma_r^j \in L_r$ where j is a number of trace.

- 1) Convert L_r to a high-level event log (denote it L_a) using data about the partition of A_r . Replace each $a_k \in \sigma_r^j$ (where k is a number of activities in the trace) to the corresponding activity from A_a for all j and k .
- 2) Use a rule introduced by us about repetitive activities in L_a . Replace every sequences of identical activities

Data: Petri net as petrinet, trace from event log as trace

Result: modified Petri net

```

trace ← sort(trace);
// index of current activity
index ← 0;
currentActivity ← trace[index];
while index < trace.size - 1 do
  indexOfNext ← index + 1;
  nextActivity ← trace[indexOfNext];
  if currentActivity == nextActivity then
    // check count of transition
    // with this name in Petri net
    if countInNet(petrinet, currentActivity) == 1
    then
      // add loop to Petri net
      // for current activity
      addLoop(petrinet, currentActivity);
    end
    indexOfNext ← indexOfNext + 1;
    while indexOfNext < trace.size and
    currentActivity == nextActivity do
      | indexOfNext ← indexOfNext + 1;
    end
    index ← indexOfNext - 1;
    currentActivity ← trace[index];
  end
end
return petrinet;
end

```

Algorithm 2: Method *addLoops()* for model's transformation by adding loops

- in σ_r^j (for all j) to one instance of this activity, i.e. $\{\{a_a^k \dots a_a^k\} \rightarrow a_a^k\}$.
- 3) If the result obtained in the previous steps (L_a) does not contain traces with repetitive activities (unlike the previous step, they are non-consecutive like $\{a_r^1, \dots, a_a^k, a_a^{k+1}, \dots, a_a^k, \dots, a_a^n\}$), then stop, otherwise proceed to the next step.
- 4) Working with each trace individually find all a_a , which have repeats in the same trace (see the previous step) and transitions in N_a , which correspond to these actions.
- 5) Add a loop to N_a for all transitions from the previous step (denote the current transition by t):
 - a) Choose one place among $p_i \in P$ and $p_i \in t^\bullet$ (denote it by p').
 - b) Add to N_a a new transition (denote it by t').
 - c) Add to N_a a new place (denote it by p'').
 - d) Add to N_a a new arcs: $\{(p', t'); (t', p''), (p'', t)\}$
- 6) Apply any known algorithm for conformance checking of L_a to N_a .

We illustrate the algorithm by applying it to the example that was presented above in Fig. 2 and 4.

First, we convert the event log L_r to a high-level log by applying Algorithm 1. The log obtained as the result of this is denoted by L_a and is shown in Fig. 6. Then we apply Algorithm 2 to the abstract model N_a and obtain the new model N'_a , shown in Fig. 7. The model N'_a is a stuttering model

over abstract activities. And finally we check conformance between the model N'_a and the log L_a by replaying traces from L_a in N'_a . It turns out, that all traces from L_a can be replayed in N'_a , i.e. the log L_a perfectly fits N'_a . This is not by chance. The following theorem states, that the proposed conformance checking method is stable under perfect fitness.

Theorem 1. Let N_r be a refinement of N_a and L_r be an event log over the set of activities A_r , i.e. $L_r \in \mathcal{M}(A_r^*)$. If L_r perfectly fits N_r , then the main algorithm return 1, which is interpreted as L_r perfectly fits N_a .

We omit the proof of the theorem, since it is rather technical and straightforward.

V. IMPLEMENTATION

The proposed method for checking conformance of high-level business model to low-level event log is implemented as a plug-in for ProM.

Our tool consists of six main classes:

- 1) *TransformerForConformanceChecking* class is responsible for interaction with framework and GUI.
- 2) *HighLevelTransition* class represents a high-level transition. Each object of this type have a name of appropriate abstract activity and an array of low-level activities, corresponding to this object.
- 3) *Activity* class represents an activity and implements forming of activity with data from event log.
- 4) *ConvertorForLowLevelLog* class is responsible for implementation of Algorithm 1, i.e. it transforms a low-level event log to an abstract event log.
- 5) *ConvertorForModel* class is responsible for implementation of Algorithm 2, i.e. it transforms an abstract model by adding the requisite loops.

VI. CONCLUSION

Abstract models are much more clear and more easily understood than low-level models. But in practice we have only low-level logs, which cannot be used for direct conformance checking. Hence checking conformance of a high-level business model to a low-level event log is an important task to facilitate the expert's work. In this paper we have presented a method for solving this problem. Also we had developed a ProM plug-in which implements the proposed algorithm.

We have proved, that our method recognizes perfect fitness between an abstract model and a low-level log correctly. This can be considered as a justification of the proposed approach. However, this is not enough. It is very important to check the method on logs with deviations. In the further research we plan experiments with different logs (logs with noise and different kinds of deviations), as well as real application logs, and we shall work on improving the algorithms through the use of found heuristics.

ACKNOWLEDGMENT

This study was carried out within the National Research University Higher School of Economics' Academic Fund.

$L = \{ \langle \text{register_request}, \text{examine_thoroughly}, \text{check_ticket}, \text{examine_thoroughly}, \text{check_ticket}, \text{examine_thoroughly}, \text{decide}, \text{reject_request} \rangle, \langle \text{register_request}, \text{examine_thoroughly}, \text{check_ticket}, \text{examine_thoroughly}, \text{check_ticket}, \text{examine_thoroughly}, \text{decide}, \text{pay_compensation} \rangle, \langle \text{register_request}, \text{check_ticket}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{decide}, \text{pay_compensation} \rangle, \langle \text{register_request}, \text{check_ticket}, \text{examine_thoroughly}, \text{check_ticket}, \text{examine_thoroughly}, \text{decide}, \text{reject_request} \rangle, \langle \text{register_request}, \text{examine_thoroughly}, \text{check_ticket}, \text{examine_thoroughly}, \text{decide}, \text{pay_compensation} \rangle, \langle \text{register_request}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{check_ticket}, \text{decide}, \text{reinitiate_request}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{decide}, \text{reject_request} \rangle, \langle \text{register_request}, \text{check_ticket}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{decide}, \text{pay_compensation} \rangle, \langle \text{register_request}, \text{check_ticket}, \text{examine_casually}, \text{check_ticket}, \text{decide}, \text{reinitiate_request}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{decide}, \text{pay_compensation} \rangle, \langle \text{register_request}, \text{check_ticket}, \text{examine_casually}, \text{check_ticket}, \text{examine_casually}, \text{decide}, \text{reject_request} \rangle \}$.

Fig. 6. The abstract event log obtained by applying Algorithm 1 to the initial event log in Fig. 4

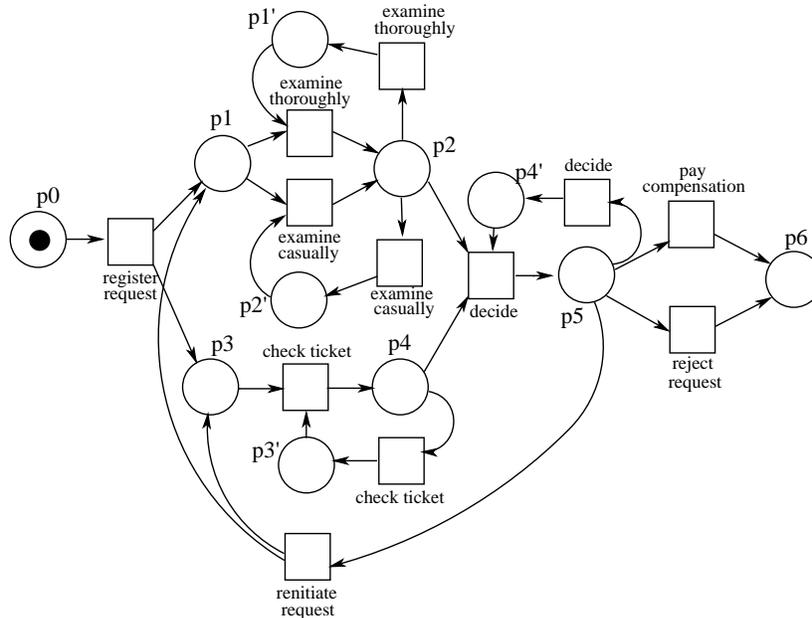


Fig. 7. The abstract model after adding loops (by applying Algorithm 2 to the model in Fig. 2)

REFERENCES

[1] W.M.P. van der Aalst. *Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, Berlin, 2011.

[2] W.M.P. van der Aalst, K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT Press, 2002.

[3] A. Rozinat, and W.M.P. van der Aalst. Conformance Testing: Measuring the Alignment Between Event Logs and Process Models. *BETA Working Paper Series*, WP 144, Eindhoven University of Technology, Eindhoven, 2005.

[4] B.F. van Dongen, A.K. Alves de Medeiros, H.M.W. Verbeek, A.J.M.M. Weijters, and W.M.P. van der Aalst. The ProM framework: A New Era in Process Mining Tool Support. Vol. 3536 of *Lecture Notes in Computer Science*, pp. 444-454, Springer, 2005.

[5] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. Prom 6: The process mining toolkit. *Proc. of BPM Demonstration Track*, vol. 615, pp. 3439, 2010.

[6] A. Rozinat. Process mining: conformance and extension. *TU Eindhoven, Diss*, Eindhoven, 2010.

[7] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance Checking Using Cost-Based Fitness Analysis. *IEEE 15th International Enterprise Distributed Object Computing Conference*, pp. 55-64, 2011.

[8] K. Jensen, and L. M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

Applying graph grammars for the generation of process models and their logs

Valeriia Kataeva
School of Software Engineering
Software Management Department
NRU – Higher School of Economics
Moscow, Russian Federation
lerileri25@gmail.com

Dr. Anna A. Kalenkova
School of Software Engineering
Software Management Department
NRU – Higher School of Economics
Moscow, Russian Federation
akalenkova@hse.ru

Abstract - This work is dedicated to one of the most urgent problems in the field of process mining. Process mining is a technique that offers plenty of methods for the discovery and analysis of business processes based on event logs. However, there is a lack of real process models and event logs, which can be used to verify the methods developed to achieve process mining goals. Hence, there is a need in an instrument that would generate process models and logs, thus allowing verification of the process mining discovery algorithms. This aim can be reached by the creation of a model and log generator.

In this paper a possible solution for the creation of such a generator will be proposed. Namely, it is the generation of process models and event logs using the rules of graph grammars on the example of structured workflow nets. The approach proposed is based on the creation of grammar rules to generate a model and an event log, which fits this model. The evaluation of the process discovery algorithms will be available due to the presence of initial models and event logs generated on the basis of these models. The tools used to perform this work are publicly available.

This paper is the research-in-progress, which is conducted in frame of master's thesis in the field of software engineering.

Keyword: process mining, discovery algorithms, conformance checking, graph rewriting rules, graph grammar, event logs.

I. INTRODUCTION

Process mining [1] is a process management technique that allows the analysis of business processes based on event logs. The basic idea is to extract knowledge from event logs recorded by an information system. Process mining aims at improving this by providing techniques and tools for discovering process, control, data, organizational, and social structures from event logs. Moreover, process mining is an approach to compare the analyzed events with preferred or predefined models or rules. The key point here is that the model has to be evaluated according to the criteria of how well it matches to the real-life process. This evaluation requires as many as it is possible event logs.

Event logs can be used to conduct three types of process mining:

- *Discovery.* A discovery algorithm takes an event log and produces a model. This can be demonstrated on the example of α -algorithm [1]. The algorithm takes an event log and produces a Petri net explaining the behavior recorded in the log.
- *Conformance checking.* In this method, an existing process model is compared with an event log (or with a model) of the same process. The conformance checking is used to check whether information recorded in the log (or in the model) corresponds to the model discovered.
- *Enhancement.* The idea here is to extend or improve existing process model using additional information about the process recorded in the event log.

The area of our research is presented in Figure 1. First, we will generate an initial model, as far as there is a lack of real examples from the business. After, we will extract logs from the model. The logs, further, will be used for applying discovery algorithms and hence a creation of a new model. Finally, the initial and a new model will serve as an input data for conformance checking.

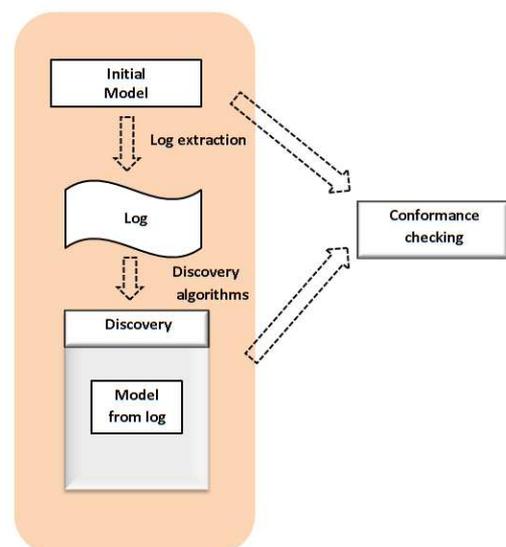


Figure 1. The graphical representation of research area.

We will start the research with the problem definition. The problem is that only a small amount of logs are available. This is caused by the fact that many industries are uncomplying to make their private data public. And this appears to be a serious obstacle for the reconstruction and developing more effective process discovery algorithms.

In this paper, we will present a possible solution to the problem stated above. The solution is based on the GROOVE – a graph transformation tool set, which allows for creating and applying graph grammars [7].

In this work we will use workflow nets (WF-nets). WF-nets are the subclass of Petri nets. A Petri net is a triple (P, T, F) :

- P is a finite set of places,
- T is a finite set of transitions, such that $P \cup T = \emptyset$,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).

A place p is called an input place of a transition t iff there exists a directed arc from p to t . Place p is called an output place of transition t iff there exists a directed arc from t to p . $\bullet t$ is used to denote the set of input places for a transition t . The notations $t \bullet$, $\bullet p$ and $p \bullet$ have similar meanings, e.g. $p \bullet$ is the set of transitions sharing p as an input place.

At any time a place contains zero or more tokens, drawn as black dots. The state, often referred to as marking, is the distribution of tokens over places, i.e. $M \in P \rightarrow N$. The number of tokens may change during the execution of the net. Transitions are the active components in a Petri net: they change the state of the net according to the following firing rule:

- (1) A transition t is said to be enabled iff each input place p of t contains at least one token.
- (2) An enabled transition may fire. If transition t fires, then t consumes one token from each input place p of t and produces one token for each output place p of t [3].

A Petri net $PN = (P, T, F)$ is a WF-net (Workflow net) iff:

- (i) PN has two special places: i and o . Place i is a source place, such that $\bullet i = \emptyset$. Place o is a sink place, such that $o \bullet = \emptyset$.
- (ii) If the transition t^* is added to PN , which connects place o with i (i.e. $\bullet t^* = \{o\}$ and $t^* \bullet = \{i\}$), then the resulting Petri net is strongly connected.

The second requirement (ii) (the Petri net extended with t^* should be strongly connected), states that for each transition t (place p) there should be directed path from place i to o via t (p). This requirement has been added to avoid dangling nodes, i.e. tasks and conditions which do not contribute to the processing of cases [1].

Business processes in the particular sphere or a company can be formalized via WF-nets, which define their semantics.

The WF-net specifies a set of tasks required to process the business cases. Also, it defines the order in which these tasks have to be executed. However, as it was already mentioned, there is an urgent lack of the models and event logs that can be analyzed according to the reluctance of the companies.

To piece out the lack of such model graph grammar rules can be applied. Graph rewriting technique is one that allows creating a new graph out of an original graph algorithmically. The definition of grammar is based on well-known process patterns, particularly in this case, patterns for WF-nets [8]. The general idea is to use the basic patterns for the generation of the process via grammar. Note that an approach for generating models using grammars was already presented in [6]. The main advantage of the approach presented in this paper is that we use an external tool to generate models, which allows working with arbitrary graph grammars. Thus, we are not bounded to the concrete processes models. Moreover, we propose an approach for a log generation based on graph grammars as well.

II. GRAPH GRAMMAR

Graph grammars are used for graphs generation. The grammar is specified by a start graph and a set of production rules. The aim of production rules is to replace one part of a graph by another (these parts of graphs are highlighted in blue and green respectively in the figures below) [4]. Moreover, as it will be seen from the examples below, each production rule is applicable under the specified conditions, which take into account the types of nodes. These conditions could be also formalized and each node can modify its attribute value according to the rules [3]. Here we would like to show an example of the generation of a structured WF-net [3], which could be defined as a hierarchy of subprocesses, based on a graph grammar. First, an initial or start graph was set and a transition counter was initialized. This is demonstrated in Figure 2.

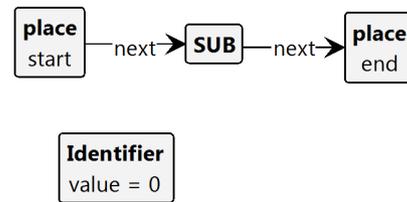


Figure 2. Start graph for applying graph grammar.

According to this image, there are two nodes of type *place* that denote the beginning and the end of the model. The node of type *SUB* (the *SUB* block) is defined as the subgraph that has to be modified according to the grammar rules. Below the graph in Figure 2 there is a node of type *Identifier* that is used for a transition identifier generation. Initially, we've put zero number.

Further, the rules applying for the graph generation were created. They contain four rules. They are:

- Transition
- Sequence

- AND-joint
- XOR-joint

R1. The rule removes the *SUB* block and sets the transition. Meanwhile, the number for the transition is incremented. The identifier for the transition is put into the newly created container which is connected with the particular, newly created transition (Figure 3).

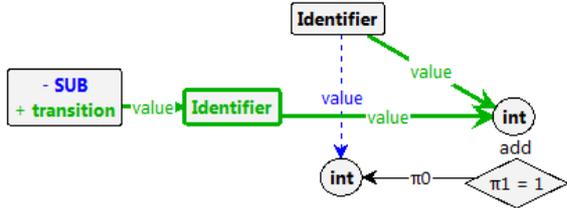


Figure 3. Rule#1 for generation WF-net.

R2. This rule replaces the *SUB* block according to the following rule that is demonstrated in Fig.2. The rule creates the sequence of nodes with the types *SUB*, *place*, *SUB*. Further, other rules can be applied to the *SUB* block.

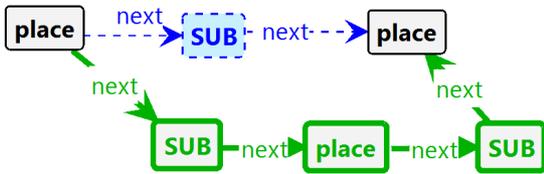


Figure 4. Rule#2 for generation WF-net.

R3. The rule is used for the replacement of *SUB* block with AND-joint combination (Figure 5).

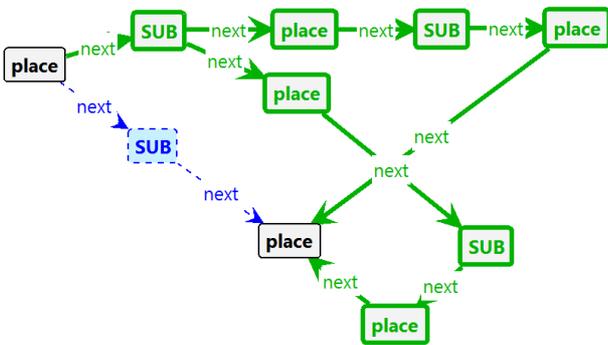


Figure 5. Rule#3 for generation WF-net.

R4. The last rule is used for the creation of OR-joint (Figure 6).

In this chapter, we have demonstrated the key principles of applying graph grammar for model creation. Note that more rules for the expansion of nodes with type *SUB* can be added, such as loop, inclusive join and others.

However, our main aim is not only about to create a model. Model is just a raw material. Further we have to extract the

execution log from this model in order to apply a discovery algorithm.

The idea of a log generation is presented in the next chapter.

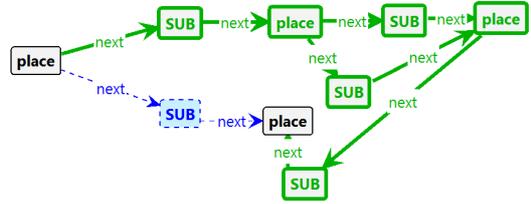


Figure 6. Rule#4 for generation WF-net.

III. LOG GENERATION

As it was already mentioned event logs allow analyzing, detecting problems and finding the solutions for process optimization.

Let A be the set of activities, which could be recorded in the event log, then the set of pairs (or records) $\sigma \subseteq A \times T$, such that there are no two pairs with the same timestamp, where T is a set of timestamps, denotes a trace. An event log $L \in \mathcal{B}(A^*)$ is a multiset of such traces.

So, every trace in a real-life event log is considered a set of event identifiers and corresponding timestamps.

After creating a model, containing no nodes, which could be expanded, as it was demonstrated above, we use this model as a start graph for the log generation. The start graph is pictured in Figure 7.

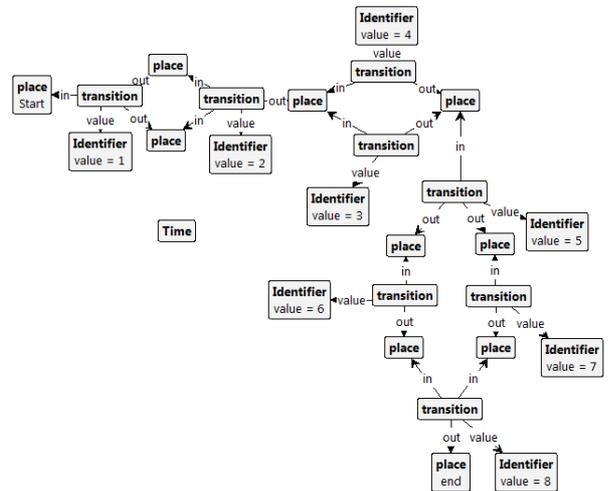


Figure 7. Start graph for the Petri net.

By moving a token through the model each trace is to be built. Turning to the same example of a WF-net, we will put a token on the start position in an attempt to keep track on what transition is executed, when we move.

In order to capture the log a time node is created, it serves as a counter, so that after the execution of every transition the value of its attribute is incremented.

For this realization the following rules were created:

- Time initialization
- Putting token on start position
- Firing rule

Time initialization. This rule as a default is executed first. It sets the value of the counter as 1 (Figure 8).



Figure 8. Time initialization rule.

Putting token on start position. The next rule puts an initial token on the start position. This is needed because the start graph does not contain it (Figure 9).



Figure 9. Initialization of token on start position.

Firing rule. This rule moves tokens according to their positions and quantity, meanwhile creating nodes with the time and identifier of the executed transition (Figure 10).

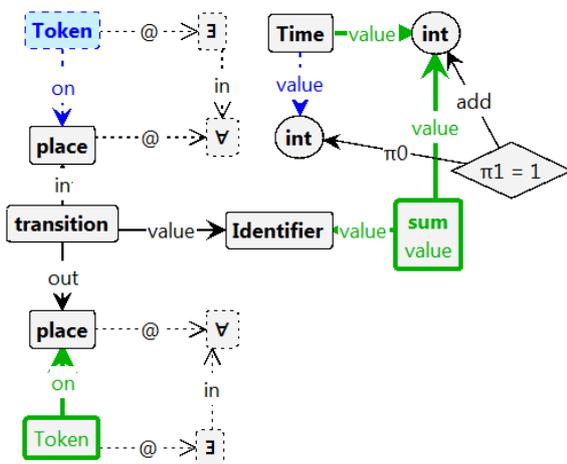


Figure 10. Firing rule for Petri Net.

After applying of firing rules and creating the final, already-executed model we can obtain the set of elements with the time and identifier of every transition fired. Namely this set can further be used as a trace.

The log created due to applying the rules has to be converted into the proper format. The proper format depends on the software that is planned to be used for the log analysis. According to this, the next chapter will describe the destination

software and give additional information on the realization of the general idea.

IV. APPLYING THE APPROACH IN FRAMES OF PROCESS MINING

In this section, we will put the main technical principles that were used for the creation of the graph grammars and the description of software. Then, we will present an idea of how these grammars can be applied for the possible solution for extracting logs [6] from the models generated and how this approach can be applied for the integration with another framework that supports a wide variety of process mining techniques in a form of plug-ins.

First we will start with the technology and software that were used for graph grammar creation. For this purpose we have chosen the tool called Groove Simulator [7].

The GROOVE tool is an instrument that is aimed on the use of simple graphs for modelling different structures of object-oriented systems and graph transformations as a basis for model transformation and operational semantics. The GROOVE tool contains:

- an editor for creating graph production rules;
- a simulator for visualization of the graph transformations;
- a generator for automatic search of state spaces and a model checker.

The GROOVE tool set was used to create WF-net generation rules, which were demonstrated above. Using the generator it became possible to produce a numerous quantity of models for the further log extraction. The WF-nets were obtained using different exploration strategies that are predefined in GROOVE and rules priorities as well. However, in the majority of cases the Random strategy was used. This strategy allows applying all the rules with an equal probability of 50%. Other strategies were tested as well in an experimental mode.

A grammar for a log generation from a given WF-net was created. The generation of a log was constructed in such a way that after the execution of every firing rule a new node, containing time and event identifier was created.

All the generated WF-nets and corresponding event logs were saved in one of the XML-formats for the further integration with ProM tool [1].

ProM is an open source Java- framework that offers a variety of process mining techniques, which are represented by the plug-ins. Currently, ProM supports import of Petri nets and event logs in specially developed XML-formats.

It is planned to implement the integration with ProM on the basis of XML-documents conversion. One possible and more probable solution for the integration is the utilization of Extensible Stylesheet Language Transformations (XSLT) - language XML-documents transformations. Based on this the plug-in for ProM that will allow importing of models and logs will be created and namely now the work is in progress.

V. CONCLUSION

The basic idea of this paper was to investigate and present the possible usage of graph grammars in solving the problem of shortage of models and logs for verifying process mining methods.

In frames of this study graph grammars for structured WF-nets and log generation were developed. We have started the research with only these types of models; however, it is important to notice that these grammars can be adapted to other more difficult and advanced process models such as causal nets, process trees and BPMN models.

Now it is possible to generate a variety of models and logs for applying discovery and further conformance checking algorithms. The generation of models can be organized via built-in extraction algorithms of GROOVE Simulator.

For the further research we are planning to develop an algorithm for conversion and a convertor itself, that will allow integrate the GROOVE models with ProM tool. The integration will be investigated due to the development of model and log import plug-in for ProM.

This paper is considered to be a part of a more complex research that will be conducted further and be expressed in the master thesis. This research will be dedicated to the development of process model and log generator based on the

appliances of graph grammars. Now it is planned to create a program that will allow defining various process models and rules for logs generation.

REFERENCES

- [1] W.M.P. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
- [2] W.M.P. van der Aalst. "Verification of Workflow Nets", *Lecture Notes in Computer Science*, vol. 1248, pp. 407-426, 1997 [Application and Theory of Petri Nets 1997].
- [3] I. A. Lomazova, I. V. Romanov. "Analyzing Compatibility of Services via Resource Conformance". *Fundamenta Informaticae*, No. 1-2, vol. 128, 2013, pp. 129-141.
- [4] F. Hermann, H. Kastenberg, T. Modica. "Towards Translating Graph Transformation Approaches by Model Transformations", *Electronic Communications of the EASST*, vol.4, 2006 [Proceedings of the Second International Workshop on Graph and Model Transformation, 2006].
- [5] A. Cavalheiro, "Relational approach of graph grammars," Ph.D. dissertation, INF, UFRGS, Brazil, 2010.
- [6] A. Burattin, A. Sperduti. "PLG: a Framework for the Generation of business Process Models and their Execution Logs", *Lecture Notes in Business Information Processing*, vol. 66, 2011, pp 214-219 [Business Process Management Workshops 2011].
- [7] A. Rensink, I. Boneva, H. Kastenberg, T. Staijen. "User Manual for the GROOVE Tool Set", Department of Computer Science, University of Twente, 2012.
- [8] N. Russell1, A. H. M. ter Hofstedel, W. M.P. van der Aalst, N.Mulyar. "Workflow control-flow patterns", *BPM Center Report BPM-06-22*, BPMcenter.org, 2006.

Generation of a Set of Event Logs with Noise

Ivan Shugurov

International Laboratory of
Process-Aware Information Systems
National Research University
Higher School of Economics
33 Kirpichnaya Str., Moscow, Russia
Email: shugurov94@gmail.com

Alexey A. Mitsyuk

International Laboratory of
Process-Aware Information Systems
National Research University
Higher School of Economics
33 Kirpichnaya Str., Moscow, Russia
Email: amitsyuk@hse.ru

Abstract—Process mining is a relatively new research area aiming to extract process models from event logs of real systems. A lot of new approaches and algorithms are developed in this field. Researchers and developers usually have a need to test and evaluate the newly constructed algorithms. In this paper we propose a new approach for generation of event logs. It serves to facilitate the process of evaluation and testing. Presented approach allows to generate event logs, and sets of event logs to support a large scale testing in a more automated manner. Another feature of the approach is a generation of event logs with noise. This feature allows to simulate real-life system execution with inefficiencies, drawbacks, and crashes. In this work we also consider other existing approaches. Their forces and weaknesses are shown. The approach presented as well as the corresponding tool can be widely used in the research and development process.

Keywords—Process mining, Petri net, event log, event log generation, ProM.

I. INTRODUCTION

In this paper we present the approach for generation of a set of event logs. This work has been done within the bigger project related to a process mining research.

Process mining is a research area which aims to discover, monitor and improve real processes by extracting knowledge from event logs available in today's information systems [1], [2].

Two main fields of process mining are: process discovery and conformance checking. *Process discovery* [3] aims to solve the following problem: Given an event log consisting of a collection of traces, construct a Petri net that adequately describes the observed behaviour [1]. *Conformance checking* [4] aims to solve the problem as follows: Given an event log and a Petri net, diagnose the differences between the observed behaviour (i.e., traces in the event log) and the modelled behaviour [1].

Process models have applications in different fields of a modern industry. Banking, insurance, software engineering, and production management are examples of such fields.

Enormous work has been done for developing the process mining algorithms. ProM tool is a framework which gathers the majority of approach implementations for process mining [5], [6]. Core part of the ProM has been developed using Java over the last years by the process mining group at the Eindhoven University of Technology. This tool is open-source and it can be downloaded from the Internet.

ProM contains a wide variety of plug-ins. However researchers are continuously inventing new and more sophisticated methods for process mining. Every new method should be tested and evaluated in different ways. The first step of evaluation for every process mining method are tests using with artificial event logs. In this work we propose the new tool which allows to generate artificial event log with defined properties.

Researchers describe the incredible growth of data [7]. Big data is a new field of research which aims to process huge amounts of data in different industry sectors. One of the main challenges of modern process mining is to turn torrents of event data (Big Data) into valuable insights related to performance and compliance [8]. A lot of work being done now explores this direction.

In order to support these research we enrich capabilities of our tool to generate sets of event logs with defined properties. This is the first main feature of our method for log generation. Another feature serves to add noise. Real data often contains noise and inefficiencies which should be filtered (or processed) by an evaluated algorithm. Researchers have a need to evaluate new algorithms using event logs containing noise with special characteristics. In this paper we propose approach for noise adding in generated event logs.

All the ideas and approaches considered in this paper are implemented as a plug-in for ProM tool. We used standard data structures and approaches accepted in ProM community [5], [6]. Thus, our implementation can be easily used and integrated.

The remainder of this work is organized as follows. In section II we analyse other works in which log generation is considered. Section III gives a description of the tool, approaches and algorithms. Section IV concludes the paper.

II. RELATED WORK

When creating new algorithms or improving the already known in the young area of process mining it is crucial to have the possibility of multiple generation of process log for a specific model. Developments in this area help researchers not only to verify concepts of algorithms but also to improve them based on model behaviour. When we provide a researcher an opportunity to manipulate a big number of behavioural examples of a model, it leads to higher quality of products being developed. Several tools have been developed to handle

this task. In this section we will take a look at existing tools for creation of event logs. We consider their main features, strengths and weaknesses.

a) *CPN Tools (see [10])*: CPN Tools is a widely used program to work with colored Petri nets. It supports the visual editing of Petri nets, simulation and analyses. This specific extension of CPN tools provides the possibility to generate random events log based on a given Petri net and produce the result log in MXML considering that the log will be used by ProM. CPN Tools has more or less usable GUI, but it is not intuitive. The main difficulty of a log creation is that it implies writing scripts in rarely used Standard ML language which leads to problems with extension of the tool and adapting it for a specific task. A user has to learn additional functional language. At the same time, the tool has a lot of applications in the field of colored Petri nets analysis and simulation.

b) *Process Log Generator (see [11])*: Process Log Generator (PLG) is a plug-in for ProM framework which enables to create random BPMN models from common workflow patterns and to simulate execution of these processes. PLG implements models customization by changing basic pattern percentages: loop percentage, single activity percentage, sequence percentage, AND split-join percentage, XOR split-join percentage. Furthermore, it gives users an opportunity to select distribution from Standard Normal, Beta and Uniform which is used to choose between random methods designated to decide which activity will be used. Noise log records can be generated throughout simulation of execution and it is possible to choose noise level. This tool is very useful for big scale brute force testing of an algorithm. The plug-in generates a set of models and an execution log for each model. Unfortunately, a user can not use existing model for logs generation. Thus, one can not make a fine adjustment of an experiment.

c) *SecSy Tool (see [12])*: Another instrument for a generation of event logs is SecSy tool. SecSy has been developed in a form of a standalone application allowing flexible settings of process models and their executions. It can create sets of logs per one run and add some deviations from the original model. The results can be produced in both MXML and XES formats. This tool is made to run experiments with security-oriented information systems. It allows to generate special event logs with particular parameters useful for security analysis of processes. Unfortunately, this orientation imposes restrictions on models which can be used by tool. Resulting event logs are also hardly useful when testing non-security-oriented algorithms.

d) *Manual generation*: Manual generation of logs has evident limitations and disadvantages including the necessity of learning XES or MXML standard. Creation of a big number of logs through manual generation is extremely tedious and inevitably leads to a tremendous quantity of mistakes. It is also very time-consuming activity even if a researcher has enough experience.

As we've seen, all of these tools have inconveniences. Using PLG user cannot use any existent models, because they are generated automatically. These tools do not provide possibility to change probabilities of outputs to be fired. None of them apart from CPN Tools do not support visual editing of models. When running some tools even a small mistake may

cause significant deviation of results and give false view about correctness of algorithms.

III. TOOL OVERVIEW

A. Functionality

In this paper we present the tool that intended to help researchers to generate sets of event logs by a Petri net replay. Petri net is a mathematical modelling language also known as a place/transition net. It is commonly used for modelling and representation of processes and systems. *Petri net* is a directed bipartite graph constructed from the following elements:

- *Transitions* signified by bars. They serve as events which may occur.
- *Places* signified by circles. They serve as conditions and connectivity elements.
- Directed *arcs* signified by arrows.

Our tool uses a Petri net as a model for event logs generation. We use this modelling language because of its prevalence among researchers who work on business process management [1]. Formally, an event log is a multiset of traces, where each trace is a sequence of events describing the life-cycle of a particular process instance [1]. Each event is a record representing some activity of a system (or model of a system). In figure 1 the example of a Petri net is shown.

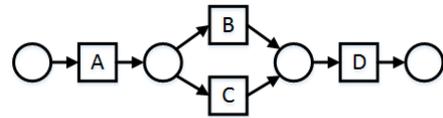


Fig. 1. An example of a Petri net

Table I shows an event log corresponding to a model from figure 1.

TABLE I. AN EXAMPLE EVENT LOG

Case	Events
0	A, B, D
1	A, B, C, D
2	A, C, D
3	A, B, D
4	A, C, D

Places in a Petri net may contain a number of tokens. Marking of a net is a distribution of tokens over the places. Marking represents a state of a Petri net. Any transition in a Petri net may fire if it is enabled, i.e. there are sufficient tokens in all of its input places. A firing of a transition is a single step in a modelled process, i.e. execution of an activity. When transition fires, it consumes tokens from input places, and produces token in its output places. Simultaneously with the transition firing an event record is added to a log. For a Petri net one can consider initial (starting) and final (ending) markings.

Process miners and developers of new algorithms for process discovering and analysis are interested in example-generation instrument. These people are core users for the

presented approach. Our tool has been developed as a plug-in for ProM 6 framework using Java 7 Standard Edition. The main features of the presented plug-in are:

- A user can easily generate a set of event logs with additional noise.
- Generation settings allow users to decide how many event logs will be generated, how many traces will these logs include. In order to prevent loops which will not terminate, user is asked about a maximum number of steps during algorithm execution. All event logs will be generated within one execution of the plug-in. By default the tool generates 5 event logs while every log consists of 10 traces and it does at most 100 steps.
- In cases when several outputs from one place are available it gives the possibility for flexible modifications of simulated behaviour which bring the higher accuracy of model behaviour describing the real world processes.
- It is possible to separate the start of a transition and the termination of a transition in event log records. Furthermore, in such cases users can define time of execution for every transition and how punctual they are executed by defining deviations bounds.
- The tool can create both event logs which completely fit the given model, and the logs with noise added.

We decided to implement our approach in the form of ProM 6 framework plug-in. The framework already has plug-ins which take care of visualization for Petri nets, event logs import and export, compatibility of logs with miner plug-ins, and provide further opportunities to work with resulting data. It was not necessary to develop additional supporting software.

B. Approach

This section describes an approach for log generation proposed in this work. Our approach contains three main parts: (1) simple log generation, (2) generation of a set of event logs, and (3) adding of an artificial noise. In the following we will consider all these parts.

1) *Generation of an event log*: This subsection describes simple log generation process. In order to generate a case in an event log the tool does the following steps:

(a) Adds tokens to all places from the initial marking.

(b) Creates an empty set which will be used to store the places with tokens. At this step only initial places have already obtained a token so they are added to the set.

(c) Next step is to select from which place we will try to fire a transition. It is handled by randomly picking a place from the set of places with tokens. Our algorithm does it without looking whether this place has outputs which could be fired or not. We do it in a way that prevents the looping in a situation when a place has a token, available outputs, but these outputs eventually lead to the same place without any other possible ways.

(d) The chosen place checks whether it has available outputs. If this is the case, an output will be chosen and fired

Data: Initial marking as `initialMarking`,
Final marking as `finalMarking`,
Settings as `settings`.

Result: Event log

```

log =  $\emptyset$ ;
time = getCurrentTime();
index = 1;
while index  $\leq$  settings.numberOfTraces do
  trace =  $\emptyset$ ;
  initialPlaces = initialMarking.getPlaces();
  finalPlaces = finalMarking.getPlaces();
  foreach place in initialPlace do
    | place.addToken();
  end
  placesWithTokens =  $\emptyset$ ;
  placesWithTokens.addAll(initialPlaces);
  step = 0;
  hasFinished = false;
  while step < settings.numberOfSteps AND NOT
  hasFinished do
    // Chooses place using random number
    currentPlace = choosePlace(placesWithTokens);
    // Tries to move from this place, if it is possible
    // moves, makes record about it in the trace
    // and returns set of places which got tokens
    newPlacesWithTokens =
    currentPlace.move(trace, time);
    removePlacesWithoutTokens(placesWithTokens);
    foreach place in finalPlaces do
      | if place.getNumberOfTokens() > 0 then
        | hasFinished = true;
        | break;
      end
    end
    placesWithTokens.addAll(newPlacesWithTokens);

    step = step + 1;
  end
  foreach place in placesWithTokens do
    | place.deleteAllTokens();
  end
  log.add(trace);
  index = index + 1;
end
return log;

```

Algorithm 1: An event log generation method

according to priorities of available outputs. Looking for the place available is done in the following way: (1) we iterate through inputs and try to hold one token from every input; (2) if we meet an input from which we cannot hold a token, it means that this output is not available; (3) otherwise, an output is available; (4) in both cases we release held tokens.

(e) Firing of a transition implies the following steps: (1) information about the event is recorded into an event log (according to the chosen settings of noise generation and timing mode); (2) tokens are added to all places which are located as outputs for this transition; (3) a set of places which got tokens is returned.

(f) Then we check if any of final places got a token. In

this case we finish the evaluation, otherwise we do the next two steps: (1) places from the original set which have no more tokens are eliminated, and (2) two sets of places are joined.

(g) Evaluation ends with deleting tokens from places which have them.

Algorithm 1 shows more precise and formal schema of the general log generation method.

Data: Initial marking as initialMarking, Final marking as finalMarking, Settings as settings.

Result: Event Log Array

$i = 0;$

eventLogArray = \emptyset ;

while $i < settings.numberOfLogs$ **do**

 log = generateLog(initialMarking, finalMarking, settings);

 eventLogArray.add(log);

$i = i + 1;$

end

return eventLogArray;

Algorithm 2: Generation of a set of event logs

2) *Generation of a set of logs:* Multiple log generation is one of the main features of the tool presented. To generate a set of logs the tool is using a loop which is called generateLog(). Every time we use it we generate one log. So we repeat it until we get the desired number of logs. If the initial marking contains several places and a set of initial places to be randomly selected, each execution of log generation method works with it's own start. A set of event logs is stored in an object of EventLogArray class. This is the special class from ProM 6 Divide and Conquer package [9] intended to store the sets of event logs. As previously mentioned, ProM framework has a modular structure and contains lots of plug-ins for different operations [6]. Several plug-ins contain classes and methods which support the work with particular modelling formalisms and approaches. In our work we use one of these common classes to work with sets of event logs. Thus, one can process the generated sets directly in other plug-ins which are based on Divide and Conquer package. Algorithm 2 is intended to generate a set of event logs.

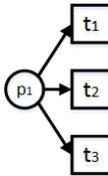


Fig. 2. Priorities

3) *Priorities:* In the case when a place has multiple output arcs the plug-in allows users to decide which output is more likely to be fired. Every output has a so-called priority which resembles the probability of this output to be selected. Each output can have the priority between 0 and 100 (including 0 and 100). Zero priority means that this output arc will be completely ignored. However, maximum priority does not mean that this output will be always fired. For every 2 outputs o_1 and o_2 it is true that relationship of the o_1 probability to

be fired to the o_2 probability is equal to the relationship of o_1 priority to o_2 priority. Hence the higher priority, the higher chance for this output to be fired. Outputs with the same priority have equal chances to be fired. Algorithm 3 shows the approach.

Lets consider the example shown in figure 2. Consider the outputs (from p_1 to t_1), (from p_1 to t_2), (from p_1 to t_3) whihc have the priorities a, b, c respectively. Plug-in creates an array with a size of 3 elements. First element is equal to a , second is equal to $a + b$, the third is equal to $a + b + c$. Then plug-in gets a random number within a range from 0 to $a + b + c$ (excluding 0 and including $a + b + c$). If this random number is less or equal to a then the output (from p_1 to t_1) is fired. If the number is bigger than a , but less or equal to $a + b$ then the output (from p_2 to t_1) is fired, otherwise the fired output is (from p_1 to t_3).

Data: List of available outputs as availableOutputs.

Result: Output transition

// Creation of array whose length is

// equal to the length of availableOutputs

priorities;

if $priorities.length > 0$ **then**

$priorities[0] = availableTransitions[0].priority;$

$i = 1;$

while $i < priorities.length$ **do**

$priorities[i] = priorities[i - 1] +$

$availableTransitions[i].priority;$

$i = i + 1;$

end

if $priorities[priorities.length - 1] = 0$ **then**

 return NULL;

end

$randomNumber = getRandomNumber(0,$

$priorities[priorities.length - 1]) + 1;$

$i = 0;$

while $i < priorities.length$ **do**

if $randomNumber \leq priorities[i]$ **then**

 return availableOutputs[i];

end

$i = i + 1;$

end

end

Algorithm 3: Selection of a transition to fire

4) *Noise adding:* Noise is defined as deliberate deviations of generated event logs from a model real behaviour. If noise is applied a user is asked to select the so-called noise level. *Noise level* shows the probability of adding noise events to a log.

In the real-life processes noise usually consists of two components. First one is a totally chaotic represents interferences or crashes. Second one has more or less strict order. This component represents breakages, incorrect or unfair activities. In this work both components are taken into account.

Noise event can be represented in several ways:

- adding artificial transitions (with names specified by user);

- adding existent transitions from a model in incorrect order;
- skipped events; in such a case artificial events and existing transitions may be added to a log.

Thus, noise is added during the log generation process. Many tools try to add noise in a totally correct event log which already generated by some instrument or obtained from any system. Another way is to change the original model and to generate correct event logs from this changed model. Our scheme is more similar to real-life process execution. We generate logs with drawbacks and deviations during process functioning, which is usual for a number of processes. Table II shows an event log corresponding to a model from figure 1 with noise added.

TABLE II. AN EVENT LOG WITH NOISE

Case	Events
0	A, B, B, D
1	A, D, B, C, D
2	A, C, D
3	A, B, D
4	A, D

C. How to use the tool

The plug-in presented has several UI screens to interact with user. We provide a number of screenshots in order to illustrate our tool and make it easier for user to begin using it. The main screen asks a user about general settings of log generation (see figure 3). User may select *desired number of logs to be created*, *number of traces per each log*, *maximum number of algorithm steps for one trace*. In case when user uses noise generation, this is not a number of events per trace: some activities may be skipped, others may be added. User may specify to the plug-in *to use (or not) priorities* and/or *noise*. The screens specifying noise generation options are shown to the user, if he (or she) selects to use generation with noise.



Fig. 3. First screen: General settings

User may specify if an execution of an activity is represented in a log with 1 or 2 events. Each transition is written as 2 log events if it is important to separate when execution of the transition starts and finishes. In such a case the first event indicates when the execution of the activity begins, whereas

the second one indicates when it ends recording information about time according to specified time of execution. In addition, if user chooses to separate the start and complete events for each activity, it is possible to set the time of execution of every activity manually or skip it and use default values.

One of the screens demonstrates to a user the Petri net given as plug-in input (see figure 4). It uses visualization plug-in from the Petrinet package to show the model. User can use this screen to specify simulation settings more simply. This is favourable for Petri nets of any size.

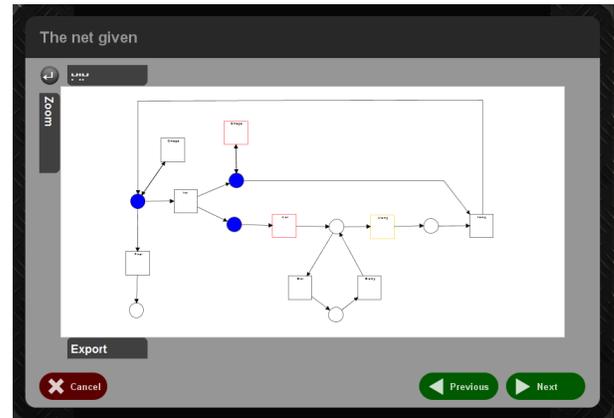


Fig. 4. Screen demonstrates a model given as plug-in input

Two screens ask user to pick an initial and final markings. User selects initial places from which an initial marking consists. The plug-in uses a final marking to end the simulation. Once a token is added to any of final places, the execution ends.

Some screens are optional. Series of screens help user to specify priorities for each place with undetermined output. Noise settings also may be specified with special screens. Users may specify the noise level and which kind of noise will be used. There are two possibilities: use only transitions of the given net, or add additional artificial transitions. Screen shown in figure 5 allows user to choose any number of transitions from a model given to appear in event log as noise transitions. Another screen helps to assign the set of artificial noise transitions. Screen with time settings allows to specify execution time for every transition (including artificial noise transitions) and maximum deviation from this time allowed for noise generator. We do not give all the screens here to not clutter up the text.

D. The tool and ProM 6 framework

This section presents an overview of the ProM 6 architecture. ProM is an open-source framework for implementation of the process mining algorithms in a standard environment. ProM consists of disjointed parts to increase the flexibility. The core part of the framework is distributed under GNU Public License. One may to upload plug-ins developed in the specific way and to work with them. The framework takes care about parameters needed for plug-ins. Special plug-ins were developed which load data into the environment and export results to disc as well as being stored in ProM resource pool for using them in other plug-ins. Almost all data types typical

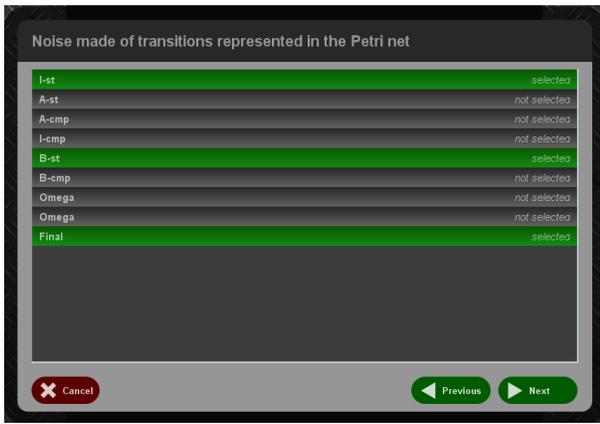


Fig. 5. Noise settings

for working with Petri nets have visualizers, so researchers and developers do not have to spend time on creating them.

Plug-ins may run with GUI or without it. It is allowed to call the other plug-ins or employ data types, visualization, import and export methods from a plug-in via special plug-in manager. The framework manages this usage in such a way: our plug-in sends a request to execute code from another package. The plug-in manager processes this request and returns an instance of the plug-in called. For example, model on screen shown in figure 4 visualized by a standard visualizer from the Petri nets package. In the future work we plan to improve this screen by using self-engineered visualizer which allows to set up several generation options (like priorities) directly on a model visualized.

Plug-in manager enables not only to call the known plug-ins but also to look for the plug-ins with specific signature, to call it, and to get results of its execution. Work with plug-ins based on such named contexts. Each plug-in must have a context. It may use the data objects within the context. For every context one can make a child context. Thus, it is possible to construct hierarchy of plug-in calls from a one parent plug-in.

The framework allows users to take an advantage of reusing previous executions of plug-ins via mechanism of so-called connections. In fact, connection is an object which holds a number of data objects in the weak hash map. Connection can be reached after its registration in a framework context by any other plug-ins using connection manager. Connection manager takes one argument and searches for all the connections which hold specific parameter. The mechanism of connections allows to process data obtained by one plug-in by another one.

The core part of a typical ProM processing plug-in is a class which contains at least one public method. This class needs to contain at least one method with special annotation which registers it in the ProM framework as a plug-in. The name, input and output parameter lists are also listed inside the annotation. Particular plug-in context of a current ProM session should be among the other parameters.

The tool which implements an approach presented in this work is built as a plug-in for the ProM Framework, therefore architecture of the tool had to fulfil all requirements of ProM

plug-ins listed above. Our tool consists of 6 main classes:

- *LogGenerator* class is responsible for interaction with framework and GUI.
- *AbstractPetriNode* represents an element of Petri net (place or transition). It wraps an object of *PetriNode* class from *PetriNets* package providing convenient access to inputs and outputs of a node.
- *Place* extends *AbstractPetriNode* getting specific features of a place. An object of this class holds a number of tokens and allows to choose between the outputs.
- *Transition* also extends *AbstractPetriNode* getting specific features of a transition. Actions of transition firing and writing to an event log are described in this class.
- *Generator* class encapsulates creation of a net acceptable for the log generation based on a given Petri net and performs generation.
- Object of *GenerationDescription* class holds information about settings specified by a user about the set of event logs to be generated (number of event logs, number of traces per log, priorities and others).

E. Example of the tool usage

Figure 6 shows several examples generated by our approach. In the first line (a), b), c)) original models are shown.

To examine our approach for each model were generated sets of event logs with different noise levels and generation settings. In figure 6 shown the model discovered by process discovery algorithm [1] from only one model for each set generated using 5% and 20% noise levels. Second line shows the models obtained using alpha algorithm [1] from the event log generated using 5% noise level. Level of 20% was used for the event logs from which the models in the third line were obtained.

In the first case (see d), g)) transitions *A - st*, *B - st* and *Final* were used as noise transitions. Transitions *e1*, *e3* and *e6* are used as noise transitions in the second case (see e), h)). In the third case (see f), i)) transitions *c* and *e* were used as noise transitions. Artificial noise transitions were used in all cases: *noise1*, *noise2*, *noise3*. Transition skipping was enabled.

We do not show the models obtained from the logs with 0% noise level. Such a model is totally identical to the original one if sufficient number of traces is used. Process discovery algorithms may show strange or inappropriate results for a tiny event logs generated from the complex models.

Resulting model complexity highly depends on structure of original model used for generation and noise setting. For example, one can choose to add lots of artificial transitions to a log. Such setting leads to a generation of an event log from which one can obtain very sophisticated model.

In our example models obtained from event logs with 5% noise level differ from original models only in several actions. Whereas in cases with 20% models are more complex and different from original ones. It is useless to generate logs using total noise levels of 50% or more: one obtains chaotic

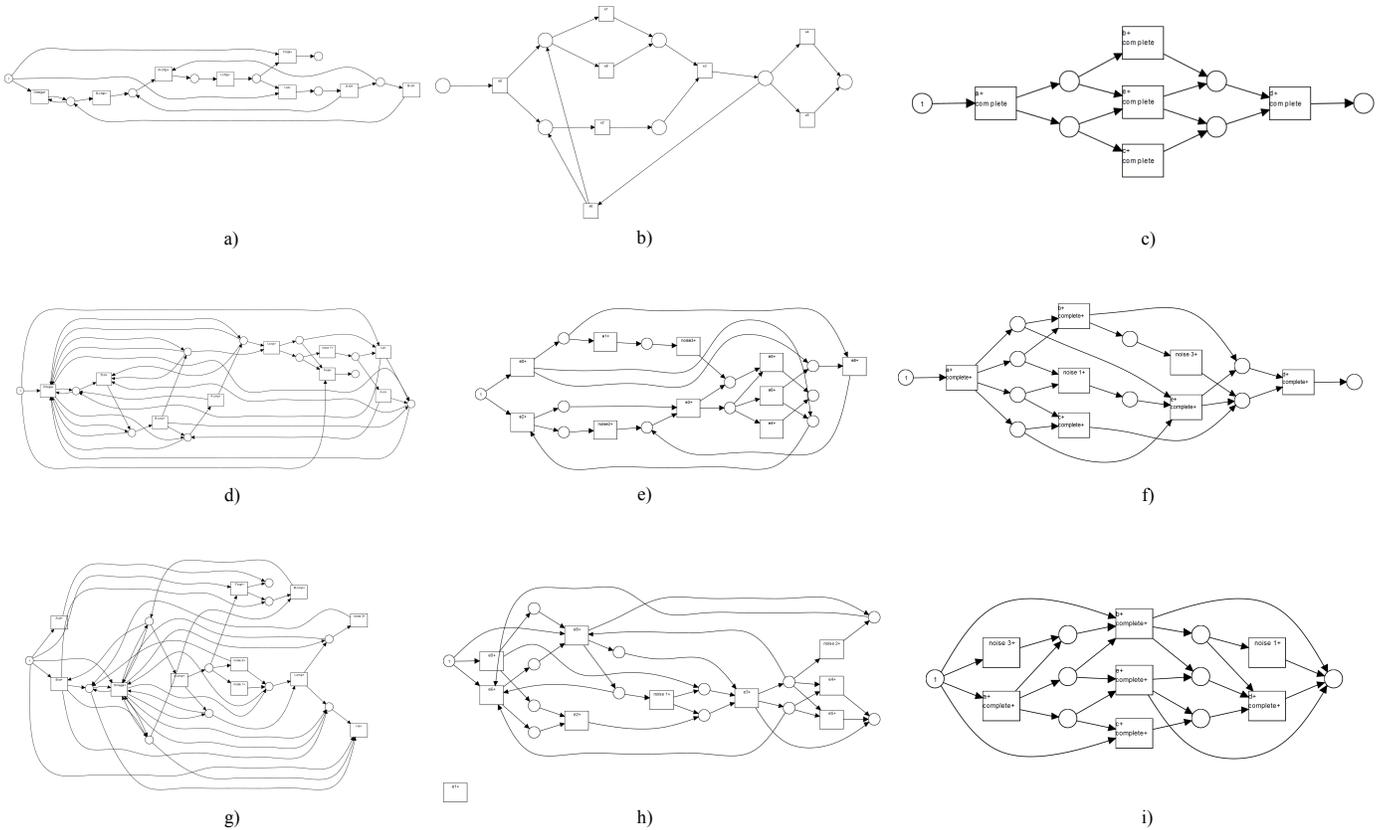


Fig. 6. Examples of models discovered from the event logs generated by the approach presented

behaviour totally different from the behaviour of an original model.

IV. CONCLUSION

In this paper we have presented an approach for sets of event logs generation. This approach is implemented as a ProM 6 framework plug-in which can be easily used by process miners, researchers, and developers. It allows not only to generate the simple event logs, but also to generate a set of event logs, or event logs with noise. All these functions allow to run experiments in the relatively easy way with different algorithms implemented as a ProM plug-ins. Generated logs can be exported using standard ProM plug-ins to use them in other applications. Noise generation is also quite useful during plug-in testing process.

The tool presented takes into account the advantages and drawbacks of other existing approaches. Nevertheless, it also has its areas to improve. In the future work authors plan to deal with a generation of logs with additional resources. Another future development is the incorporation of different model formalisms into existing plug-in in addition to the Petri nets. Several improvement should be done in graphical user interface to simplify interaction with plug-in.

ACKNOWLEDGMENT

This work is output of a research project implemented as part of the Basic Research Program at the National Research

University Higher School of Economics (HSE). Authors would like to thank all the colleagues from the PAIS Lab whose advice was very helpful in the preparation of this work.

REFERENCES

- [1] Wil M. P. van der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011.
- [2] IEEE Task Force on Process Mining, "Process mining manifesto," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, F. Daniel, K. Barkaoui, and S. Dustdar, Eds., vol. 99. Springer-Verlag, Berlin, 2012, pp. 169–194.
- [3] W. M. P. v. d. Aalst, A. J. M. M. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.
- [4] A. Rozinat and W. M. P. v. d. Aalst, "Conformance testing: Measuring the fit and appropriateness of event logs and process models," in *Business Process Management Workshops*. Springer, 2006, pp. 163–176.
- [5] B. F. van Dongen, W. M. P. van der Aalst, C. W. Günther, A. Rozinat, E. Verbeek, and T. Weijters, "ProM: the process mining toolkit," in *Business Process Management Demonstration Track (BPM Demos 2009)*, ser. CEUR Workshop Proceedings, A. K. A. d. Medeiros and B. Weber, Eds., vol. 489. Ulm, Germany: CEUR-WS.org, 2009, pp. 1–4.
- [6] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Prom 6: The process mining toolkit," *Proc. of BPM Demonstration Track*, vol. 615, pp. 34–39, 2010.
- [7] W. M. P. van der Aalst, "Decomposing petri nets for process mining: A generic approach," *Distributed and Parallel Databases*, vol. 31, no. 4, pp. 471–507, 2013.

- [8] W. M. P. van der Aalst, "Mine your own business: Using process mining to turn big data into real value," in *Proceedings of the 21st European Conference on Information Systems (ECIS 2013)*. Utrecht, The Netherlands: AIS Electronic Library, 2013, pp. 1–9.
- [9] E. Verbeek and W. M. P. v. d. Aalst, "Decomposing replay problems: A case study," in *Joint Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE'13) and the International Workshop on Modeling and Business Environments (ModBE'13)*, Milano, Italy, June 24 - 25, 2013, ser. CEUR Workshop Proceedings, D. Moldt, Ed., vol. 989. CEUR-WS.org, 2013, pp. 219–235. [Online]. Available: <http://ceur-ws.org/Vol-989/paper07.pdf>
- [10] A. K. A. d. Medeiros and C. W. Günther, "Process mining: Using CPN tools to create test logs for mining algorithms," in *Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, ser. DAIMI, K. Jensen, Ed., vol. 576. Aarhus, Denmark: University of Aarhus, 2005, pp. 177–190.
- [11] A. Burattin and A. Sperduti, "PLG: a framework for the generation of business process models and their execution logs," in *BPM 2010 Workshops, Proceedings of the Sixth Workshop on Business Process Intelligence (BPI2010)*, ser. Lecture Notes in Business Information Processing, J. Su and M. z. Muehlen, Eds., vol. 66. Springer-Verlag, Berlin, 2011.
- [12] T. Stocker and R. Accorsi, "Secsy: Security-aware synthesis of process event logs," in *Proceedings of the 5th International Workshop on Enterprise Modelling and Information Systems Architectures*, St. Gallen, Switzerland, 2013.

DPMine/C: C++ Library and Graphical Frontend for DPMine Workflow Language

Sergey Shershakov
International Laboratory
of Process-Aware Information Systems (PAIS Lab)
National Research University Higher School of Economics
Moscow 101000, Russia
Email: sshershakov@hse.ru

Abstract—DPMine generic purpose workflow language is rooted in DPMine/P scientific workflow language and a set of plug-ins for ProM which originally were developed for convenient piping of different plug-ins within ProM framework. DPMine/C is a new version of DPMine workflow language and a C++ library. The main language concept was complemented by comprehensive analysis of DPMine/C model execution semantics. This paper also discusses approaches to the *block types extension* concept relying on development of new block type classes and customization of the *model storage subsystem*. Finally, we show an approach for implementation of a GUI frontend.

Keywords: Workflow, Modelling Language, C++ Library, Extensible Tool, Process Mining, Processes, PAIS

I. INTRODUCTION

Today there exists a wide variety of workflow notations. Some of them have a formal basis (such as Petri nets, finite state automata), others have vendor specific notations. Among them one can distinguish some notations that pretend to be industry standards. BPMN [1] and BPEL [2] are, perhaps, just the most known examples of such standards [3].

At the same time, being a standard does not mean being appropriate for description of all kinds of workflow models. As an example, we refer to the papers [4], [5] where a problem of piping components (plug-ins) of ProM heterogenous system arises. We had to obtain an ability to create a *scientific workflow model* that includes individual invocations of particular processing algorithms (implemented, for example, with the help of ProM plug-ins), cycles support, choice constructs and other controls of the execution thread. As a result, DPMINE/P language with a simple, transparent, flexible and, most importantly, extensible semantics has been developed and implemented as a set of ProM plug-ins [5].

Typically, workflow management systems are used for maintaining various kinds of *business* processes. DPMine is not another Business Process Management (BPM) system. We rather call DPMine as a “technical workflow language”.

Unlike most workflow languages DPMine has much more imperative rather than declarative nature. A model in DPMine language is similar to a program in some way. As well as a program, a model can be *executed*, thus we pay much attention to its execution semantics. We explicitly state that it is “execution”, not just a “simulation”. In the case of a well-formed algorithm the model execution “outcomes” are

determined only by the model incoming *resources* and by the model structure (depending, of course, on the nature of the blocks contained in the model).

Along with the differences DPMine has also a lot of similarities with existing workflow languages. Thus, DPMine uses *ports* notations, just as BPEL (Web Services Business Process Execution Language, WS-BPEL) [6] does. But in DPMine the *port* is one of the main language building elements widely used for setting relations between *blocks* — another important building element.

There is a specific language family of Petri net based workflow languages [7]. One of the most known is YAWL — a workflow language extended with additional features to facilitate the modelling of complex workflows [8]. Unlike YAWL, DPMine does not introduce a generic set of tasks¹ that support control-flow tasks (such as AND/XOR split/join and so on) as a part of the language core. We proposed instead some control-flow blocks [5]. We considered them only as an example of custom block type implementation. In this paper we are primarily considering blocks as abstract objects.

Basically, the requirements for DPMine language have been indicated through the requirements imposed on a scientific workflow language for ProM tools. Moreover, one can say that DPMine/C has been emerging as a way of generalizing the solution of the piping task. One could consider incorporating one of the existing BPEL engines into ProM, but this approach is fraught with compatibility problems and leads to the use of BPEL in an area not much related to it. A similar situation happens with BPMN, but in this case the language application would be even more harder.

Workflow notations can be considered from different perspectives [9]. Here, we are mainly focusing on the *control-flow perspective*.

The rest of this paper is organized as follows. Section II describes the modular concept and the basic components of DPMine language. Section III discusses DPMine model execution semantics and approaches to resources transferring. Certain aspects of the extendable storage subsystem are presented in Section IV. Section V introduces an idea of implementation of a GUI frontend for DPMine language using

¹They could be treated as so-called *task-blocks* in terms of DPMine.

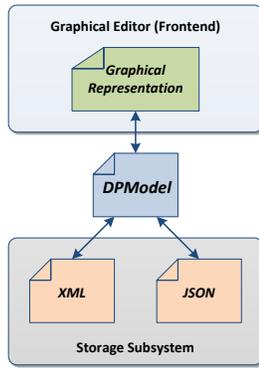


Figure 1. Levels of language representation

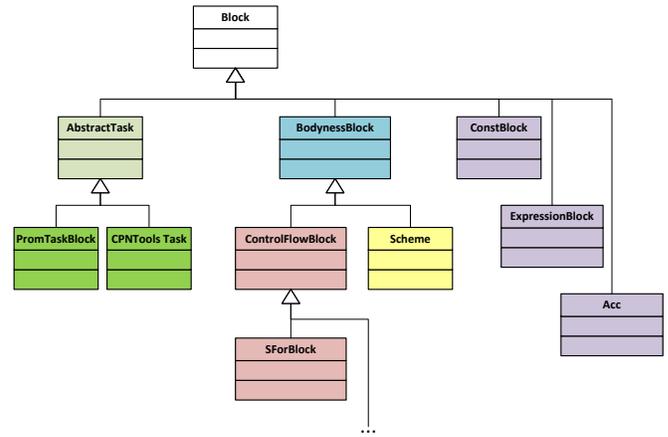


Figure 2. Blocks hierarchy scheme

Qt library. Finally, Section VI concludes with analysis of the work done and a look at the future.

II. DPMINE LANGUAGE BASIC ELEMENTS

The main work concept for DPMine/C toolset is the *model*, which represents some *workflow model under experiment*. In a C++-based types system a *model* is represented as an object of DPMoDel base class containing all necessary information about the model such as model name, model author, and so on. The most important object contained in the model is the so-called *main scheme* to be executed during the model execution procedure (see sec. III). Developing a DPMine/C library client one can extend the DPMoDel definition by adding project-relevant features.

The rest of this section is devoted to the model concept, its main components and their interaction.

A. Model Definition

A model can be considered from different points of view (Fig. 1). At the lower level there is a C++-based *object model* of the workflow model. At the medium level (it can also be referred to as a *storage level*) there is an XML-based model markup language or something similar such as JSON-based text format. Finally, at the upper level a model can be represented using a graphical notation, which allows defining the process model as a set of building blocks.

An object model can be serialized to an underlying XML- or JSON-based text file or deserialized from it (see. sec. IV). One can extend a serialization mechanism to utilize any other storage formats.

The graphical model is transformed into an object model and vice versa, and it can be used both for user-friendly model design and representing the model execution dynamic process.

B. Schemes, Blocks, Ports, and Connections Concept

Implementation of the basic language semantics is done through the concept of schemes, blocks, ports and connectors. Expansion of the language functionality should be based on this very concept. The main idea is that no special extensions for maintaining any kind of custom blocks are needed.

Let us examine these elements in more detail.

1) *Block*: It is a basic language building element considered as a solitary operation in an external representation but can be complex in internal one. Blocks perform a specific task and can be considered as statements in programming languages. Blocks can have different functionality such as performing a single task of a base platform (*task blocks*), representing complex schemes into single blocks (*scheme blocks*), implementing control workflow (*control flow blocks*) and so on.

The blocks are arranged by types into a hierarchy that is a reflection of corresponding C++ block types hierarchy, whose scheme is presented on Fig. 2.

2) *Port*: It is an object belonging to a certain block and used for connection and data objects transfer to other ports. Depending on data flow direction, one can distinguish three types of ports: *input*, *output*, and so-called *proxy* (input-output and output-input). Ports transfer *resources* of a specific data type from one block to another. Depending on block type, they can be either custom or built-in.

3) *Connector*: It is an object connecting two blocks through their ports. Connectors have a *link direction*: a connector (with its beginning) always connects an output port of a block with an input port of another one (with its end). One output port can be linked to several connectors, whereas one input port can have only one connector linked (Fig. 3a).

Starting from this implementation of DPMine library it was decided that it was not necessary to introduce a special data type representing connector at the programming level. Instead, we decided to use internal links between corresponding output and input ports. Such links can be treated as connectors at higher levels.

4) *Scheme*: It represents a number of interacting blocks connected with each other by connectors. The schemes are the main mechanism of implementing abstraction, isolation and hierarchy of sub-processes. On Fig. 3b there is depicted a connected scheme consisting of four blocks (A, B, C, D) and four connectors (AB, AC, AD, BD).

One needs to implement a special container to represent a scheme at the object model level. Such a container is a

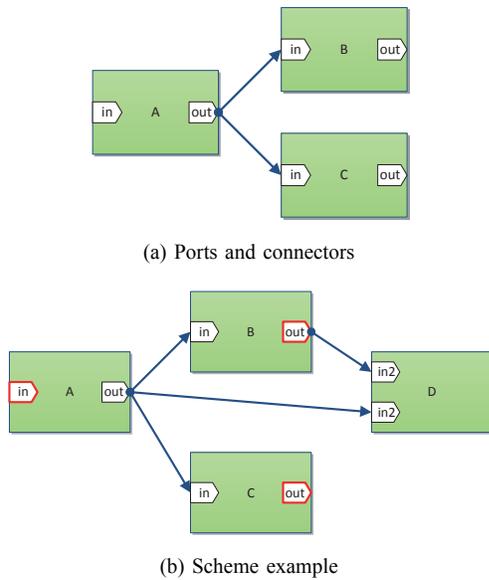


Figure 3. Various blocks, ports, connectors, and schemes elements

special type of block that is called *Scheme-block* (Fig. 2). The *Scheme-block* is a direct (and most obvious) descendant of a more general block type, namely *Bodyness block*. A *Bodyness block* is a special kind of blocks whose distinguishing feature is the ability to aggregate their so-called child-blocks inside themselves. In other words, the *Bodyness block* represented by `BodynessBlock` abstract class is the ancestor for all the block types that can own child blocks.

Any descendant of a *Bodyness block* (including *Scheme-blocks*) can be considered both at the external and the internal level. At the external level a *Bodyness block* is nothing more than just a regular block, which can have input and output ports that can be connected to some ports of some other blocks (at the same level). At the internal level the same block can be treated as an isolated scheme (maybe having some special behavior based on individual characteristics of the specific descendant²).

This scheme has the only way to communicate with external blocks at a higher level: by using its own port “everted” and represented at the internal level in the opposite direction. Thus, the ports viewed at the external level as input flip to be output at the internal level, and vice versa. This is why these ports are named “proxy”.

III. MODEL EXECUTION

One of the main goal for constructing a DPMine model is its subsequent execution. Outcome results of the executed model is the sum of results of its individual executed blocks. They are based on the subject domain of the task blocks contained in the model.

One can consider an example of workflow model containing some tasks that perform phased processing of a set of source

²For example, consider *For-loop block*, see [5].

tex-files with a view to obtaining a PDF-document. This produced PDF is such an outcome result.

Model execution consists in executing the *main scheme* of the model (upper level scheme) and producing an execution report (about errors, etc.). Model execution is done by a special agent — *Executor*, whose implementation is closely related to the client application design (see sec. III-C).

A. Block Execution

When considering the execution of a scheme one needs to mention the execution concept for an individual block. *Block execution* is an operation done by the underlying block’s type class method `execute()` based on the block’s individual state. In the example above there could be a special block type performing invocation of some LaTeX tool like `pdftolatex` as its execution procedure.

From the technical point of view `execute()` method is virtual, which means that it must be implemented individually for each type of blocks. It is also possible to modify the behavior of any previously defined block type by reimplementing this virtual method.

In order for a given block to be able to be executed it is necessary that all the external dependencies of the block be satisfied.

For a given block B its dependencies are considered satisfied if:

- 1) the block does not have input ports, or
- 2) the block has input ports and for each port the following conditions are met:
 - a) there is no “must be connected” flag for the port set, this way the port can be not connected by a connector to another (output) port of another block;
 - b) the input port is connected by a connector to another (output) port of another block and this output port is ready to give requested resource to the input port; in most cases, the latter means the status of parent block of the output port is “executed”.

The block with satisfied dependencies is referred to as “executable” block.

B. Scheme Execution

As it was mentioned above every scheme is represented by its *Scheme-block*. So, speaking about the execution of a scheme one needs to consider the execution of its block.

Since the *Scheme-block* is a descendant of the *Bodyness block*, it does not define its own execution algorithm but utilizes an algorithm given by *Bodyness block* class³.

In fact, this algorithm is the heart of DPMine execution semantics, so it has to be discussed in greater detail.

The mentioned algorithm is iterative. Some subset of the full set of scheme body blocks is tried to be executed in each iteration. During its execution the algorithm defines some

³This also holds for many other *Bodyness block* descendants like *For-loop block*, etc.

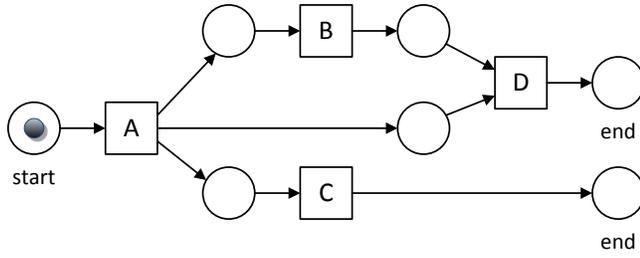


Figure 4. Equivalent (system) Petri net for the scheme on Fig. 3b

state flags. The first is *incomplete* flag indicating whether there are still some blocks that have not been executed. Then, *hasExecution* flag indicates that there is at least one body block that has been executed during the whole iteration. Finally, *hasPending* flag indicates that there are some blocks being executed at the moment.

One has to mention that the latter flag can be set only in case of assigning a task for execution in asynchronous mode with multiple concurrent executing threads. Assignment of blocks for execution is done by a special component — *Executor*, which determines a strategy of forming such assignments (see sec. III-C).

There are some steps performed by the executing algorithm in each iteration.

- 1) All three flags are set to false state indicating that no information about state of blocks-to-execute is available yet.
- 2) An effort to execute a body block is applied to each block contained in the scheme body:
 - a) It checks whether the block has already been executed previously. If so, it simply goes to the next block.
 - b) It checks whether the block is being executed (in pending state). If so, *hasPending* flag is set, and it goes to the next block.
 - c) Finally, it checks whether the block can be executed (has executable state). If so, there is a request for *Executor* to execute the current block (see sec. III-C).
- 3) If there is at least one block with a state that is different from “executed” (e.g. a block could either be not started at all or be started and still being executed) — that is *incomplete* flag is set up, and there has been at least one block execution, the next iteration is performed.

The presence of some pending blocks with no block executed is another variation of this case.

The semantics of scheme execution can be represented by an equivalent *system net* [10]. Thus, for the scheme on Fig. 3b containing no choice blocks the equivalent (system) Petri net would appear as on Fig. 4.

C. Model Executor

Model Executor (or just *Executor*) is another important component of the executing subsystem. It is a special agent

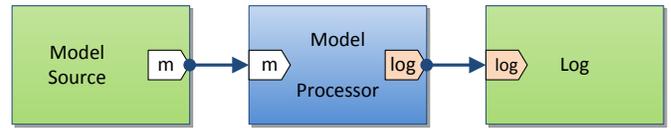


Figure 5. Pull (white) and push (yellow) ports

linking DPMine workflow model and client software together. Technically, an *Executor* is an object of some class implementing *IExecutor* interface which declares some methods used for executing individual blocks of a workflow model.

Among these methods one can distinguish a couple of the most important ones. The `executeBlock()` method is invoked by some blocks of some special types such as *Bodyness block* whenever there is another block to be executed. The block is passed to `executeBlock()` as a parameter, and the method should be considered as a request from a model to the *Executor* for executing another model’s block. The way the *Executor* performs the request is completely determined by the policy implemented in each specific *Executor* class. As an example, one can consider synchronous model of execution. In this case control is not returned to the calling method until the current block is executed or switched to a broken state. Another scenario involves asynchronous tasks assignment to different concurrent threads.

The very first call to `executeBlock()` procedure is performed when the model is being executed, and its *Main Scheme* becomes the very first block passed as a parameter to `executeBlock()`.

Another important method to be implemented by a specific *Executor* class is `execNotify()`. It is used to send to a client application event notifications about the state of a model block being executed. The pointer to the block being processed is the first parameter of the method, and the event notification type is the second. Notification type is a constant from a predefined set including *begin*, *end*, *cancel*, and others. This notification callbacks may be used by the client application for updating its information about the blocks’ states.

D. Resource Transferring

Now let us consider the process of transferring resources through the ports of blocks. We distinguish two different approaches for resources transferring: *pulling* and *pushing*. *Pulling* is an approach where a block being executed requests all necessary resources from its input ports which are connected to the corresponding output ports of other “source” blocks. As it was mentioned before, the ability to supply resources for an ahead standing block by the “source” blocks is the main requirement for the current block to be executed. Then, the request for the input ports is redirected to the corresponding output ports through established connectors. Finally, the output ports ask their owner blocks to supply data for the requested resources and give them to the requesting block.

Pushing is another approach illustrated on the Fig. 5. There is a Log block having connected to a processing

ModelProcessor block. If executed ModelProcessor generates some log entries (events) and tries to “send” them synchronously through a dedicated output port to all the recipients connected. If a receiving block is ready to get such a “message” it could process it. In the example, the Log block obtains messages from ModelProcessor for displaying and storing.

IV. MODEL STORAGE

In the preceding sections we looked at DPMine workflow models from the object model point of view. Now let us consider how a model can be represented as some formal definition in a text-based format.

In the paper [5] we have shown that a model of process mining experiment can be described by using a well-formed XML-based notation. Moreover, this kind of model description was the main form used for importing DPMModel objects in ProM.

Having started developing this library we decided to separate the storage subsystem from the core of DPMine library. According to Fig. 1 the storage engine is used for the transformation of an object model to some persistent form such as a text-based file (serialization) and back (deserialization).

As previously, we suppose an XML-based format is one of the best notations for representing a hierarchical irregular structure, which a DPMine model is. In addition, we are now proposing another well-known format with the same XML expression but also which is much more compact and, what is more important, much more appropriate for manual writing: JSON [11].

The main idea is that, as much as the DPMine core system can be extended by developing new block type classes, the storage subsystem can be simultaneously extended to maintain the core extension mirrored. For this very purpose we introduce two special classes: XMLModelLoader and JSONModelLoader. Both of these classes have methods for working with text streams. These streams are used to serialize to and deserialize from a given object model. Some of specific descendants of these classes specify whether a stream is implemented as a file-stream or as another kind of stream.

By processing a file to be loaded as a workflow object model XModelLoader parses the file standard header containing the model description and the model body containing the workflow itself. At a higher level, the model body normally contains only one Scheme-block corresponding to the main scheme. It means the parsing process should start from parsing this very block.

The extensible nature of (de)serialization mechanism consists in the fact that processing each type of blocks is performed by its dedicated block loader, custom block loaders for custom block types being added dynamically. The collection of all types of loaders including the standard ones (like Scheme-block loader based on bodyness block loader, const block loader, etc.) is managed by a family of so-called LoadersFactory classes existing for each branch of persistent formats: XML, JSON, and so on (Fig. 6).

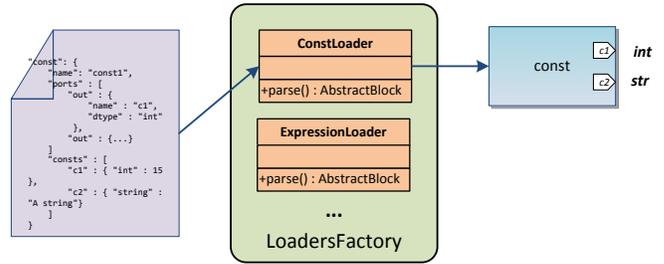


Figure 6. Const loader used for deserialization a JSON-defined “const” block

Registering a new block loader for a corresponding block type is done during the library initialization process (normally only once) that uses the underlying collection, which maps the block type name as it appears in the file (as an XML node or a JSON parameter) to a parsing object (normally it is a pointer to a method). When a file loader meets another block description (an XML node or a JSON parameter) it tries to find an appropriate block loader and eventually invokes this block loader and passes to it the cursor to a file position the block description start is located at. After that, the block loader performs the reading of all the necessary data from the stream, constructs a new block object and returns it.

V. GRAPHICAL FRONTEND

Along with the library we supply a GUI application demonstrating the ability to integrate DPMine/C library into a GUI client. The application is based on cross-platform Qt library allowing to consider the application as cross-platform. Nevertheless, we are focusing here on a Windows edition to be specific.

The topics to be considered are:

- 1) How to visualize a workflow model?
- 2) How to enable user to interact with individual schemes?
- 3) How to deal with the fact a model can contain custom block types?
- 4) How to use different look-and-feels?

A. Qt Graphics View Framework

We use Qt Graphics View Framework to make the graphical part of the application. The main components of the framework are the following.

- 1) QGraphicsView object provides a widget for displaying the contents of a graphic scheme implemented by QGraphicsScene descendant.
- 2) DPMScheme inherits QGraphicsScene class and adds functionality to handle DPMModel specific graphical items in addition to the items handled by its super class.
- 3) QGraphicsItem is an abstract class for a family of classes representing flowchart shapes — main and miscellaneous.

Flowchart shapes implemented by QGraphicsItem descendants are placed on DPMScheme object, and the latter is visualized by a QGraphicsView container. The goal is to

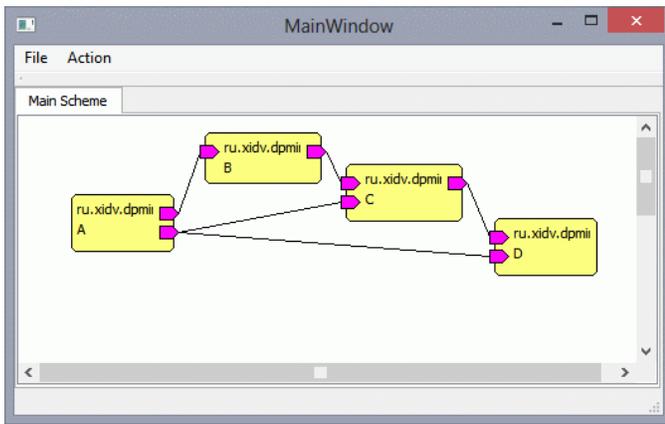


Figure 7. GUI demo application

provide an appropriate graphical renderer for specific block types taking into account the ability to change easily its graphical representation just by changing the renderer. This is the so-called look-and-feel feature.

B. Custom Block Renderers

Just as in the case of custom *block loaders* we propose a dynamic extensible mechanism of *block renderers*. `BlocksRenderers` is the main class supporting a collection of *block renderers*. It contains methods for adding and getting a renderer for a block type given by its type name. Method `getRendererByBlockTypeName` returns an appropriate `BlockTypeRenderer` object used for rendering a given block. If no appropriate `BlockTypeRenderer` for a given block type is found, a default `BlockTypeRenderer` is returned.

`BlockTypeRenderer` is the base abstract class for all the *block renderer* classes. The main method they have to implement is `renderBlock`. It returns a `QGraphicsItem` specifically representing a given block type. This representation can take into account any necessary graphical aspects of a block the developer would like to implement. The base implementation `DefBlocksRender` that can be used for all the block types renders a given block using `DPMDefBlockItem` (a descendant of `QGraphicsItem`). It only shows the block's name and its type as a text label and, of course, renders its ports (Fig. 7).

Block ports are also presented as separate objects of `QGraphicsItem` descendant class (`DPMDefPortItem` is default) grouped by the owner block shape. This is done in order to enable the user to communicate with ports as individual objects.

The presented graphical solution is one of the significant plug-ins for “VTMine framework” (under development) [12].

VI. CONCLUSION

In this paper we discussed DPMine workflow language and its implementation as a C++ based library. We introduced DPMine main concept and looked at its building elements.

Semantics of the model execution has been presented in detail. DPMine block extension approach has been mentioned with regard to the addition of new block types and extension of the storage subsystem and a graphical frontend.

Among the challenges for the future a number of tasks can be identified, namely forming a strong formal semantic system, extending the functionality of DPMine language by introducing some default block types and presenting more complex workflow use cases.

Finally, we have launched a web-site for a DPMine project: <https://prj.xiart.ru/projects/dpmine>. It is based on a Redmine bug-tracking system and we consider it as a platform for the future DPMine development.

ACKNOWLEDGMENT

The study was implemented in the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

REFERENCES

- [1] O. M. G. (OMG), “Business process model and notation (BPMN) version 2.0,” Tech. Rep., Jan 2011. [Online]. Available: <http://taval.de/publications/BPMN20>
- [2] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guzar, N. Kartha, C. Liu, R. Khalaf, D. Koenig, M. Marin, V. Mehta, S. Thatte, D. Rijn, P. Yendluri, and A. Yiu, “Web Services Business Process Execution Language Version 2.0 (OASIS Standard),” WS-BPEL TC OASIS, <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>, 2007.
- [3] W. M. P. van der Aalst, “What makes a good process model? - lessons learned from process mining,” *Software and System Modeling*, vol. 11, no. 4, pp. 557–569, 2012. [Online]. Available: <http://dblp.uni-trier.de/db/journals/sosym/sosym11.html#Aalst12>
- [4] S. Shershakov, “DPMine: modeling and process mining tool,” in *Proceedings of the 7th Spring/Summer Young Researchers’ Colloquium on Software Engineering, SYRCoSE 2013*, 2013.
- [5] —, “DPMine/P: modeling and process mining language and ProM plug-ins,” in *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, A. N. Terekhov and M. Tsepkov, Eds. ACM New York, NY, USA, 2013. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2556622&CFID=415147702&CFTOKEN=35395117>
- [6] *Web Services Business Process Execution Language Version 2.0*, OASIS Std.
- [7] W. M. P. van der Aalst, “Three good reasons for using a Petri-net-based workflow management system,” in *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC’96)*, Cambridge, Massachusetts, Nov. 14–15, 1996, Navathe, S. and Wakayama, T., Eds., 1996, pp. 179–201.
- [8] W. M. P. van der Aalst and A. H. M. ter Hofstede, “YAWL: Yet another workflow language,” *Inf. Syst.*, vol. 30, no. 4, pp. 245–275, Jun. 2005. [Online]. Available: <http://dx.doi.org/10.1016/j.is.2004.02.002>
- [9] S. Jablonski and C. Bussler, *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996.
- [10] W. M. P. van der Aalst, “Decomposing Petri nets for process mining. A generic approach,” 2012.
- [11] The application/json media type for JavaScript object notation (JSON). [Online]. Available: <http://tools.ietf.org/html/rfc4627>
- [12] P. Kim, O. Bulanov, and S. Shershakov, “Component-based VTMine/C framework: Not only modelling,” in *Proceedings of the 8th Spring/Summer Young Researchers’ Colloquium on Software Engineering, SYRCoSE 2014*, 2014, in press.

Component-based VTMine/C Framework: Not Only Modelling

Polina Kim

Department of Applied Mathematics
and Information Science
National Research University
Higher School of Economics
Moscow 101000, Russia
Email: pvkim@edu.hse.ru

Oleg Bulanov

Department of Applied Mathematics
and Information Science
National Research University
Higher School of Economics
Moscow 101000, Russia
Email: ovbulanov@edu.hse.ru

Scientific Advisor: Sergey Shershakov

International Laboratory
of Process-Aware Information Systems
(PAIS Lab)
National Research University
Higher School of Economics
Moscow 101000, Russia
Email: sshershakov@hse.ru

Abstract—The paper discussed a concept of VTMine/C — yet another modular framework which is extensible by plug-ins. The subject area of the core program refers to process modelling, workflow, and process mining fields. The framework allows third-party components in the form of plug-ins to extend the program and customize their interaction with each other and with the framework modules. An abstraction layer build up of the framework modules is described. Concepts of resources, plug-in dependencies and other features are introduced. One of the main VTMine/C purposes is to support integration with DPMine/C library.

Keywords: Component-based Application, Software Architecture, Modelling Tool, C++, Qt

I. INTRODUCTION

Nowadays it is a great challenge to present an application that can totally fulfil the user's requirements. The task is to create software that can be easily enhanced. While traditional software engineering design cannot tackle such a task, a component-based approach seems to be an appropriate solution.

This approach has diverse objectives [1] and provides a vast range of benefits like reusable code (components can be used independently or with any other environment), team development and extensibility (a framework can be dynamically extended by plugging in new components that can be renewed or released without changing the core platform).

The main goal of our work is to present a modular framework extensible by plug-ins. Plug-ins allow customizing the core functionality in a wide range. Specialization of the framework is related to process modelling, workflow, and process mining areas [2] (Fig. 1). One of the key features of VTMine/C are mathematical models (graphs and Petri nets) that are widely spread in process mining discipline.

The ability to be extended by third-party components is highly valued almost in all kinds of software. There are available different frameworks both for solving a certain type of problems and for building tools for various tasks. *Eclipse* is one of the most well-known modular and scalable cross-platform frameworks [3]. One of the most important concepts of Eclipse is that everything is a plug-in: each subsystem in the platform is itself structured as a set of plug-ins, implementing

some key functions. There are two ways how plug-ins can interact with each other: by dependencies or extensions and extension points. They are implemented to avoid tight connections between components.

Another example is related to Data Mining area. *RapidMiner* software supports environment for predictive analytic tools, data mining tools, etc. Developers are provided with a platform to create custom plug-ins and implementations of various mining algorithms. Furthermore, developers are provided with hooks to change RapidMiner's behaviour [4].

Such systems as Mirand-IM [5], Mozilla Firefox [6], Hudson [7] and Notepad++ [8] are also built up of plug-ins coupled together. One can easily extend their possibilities by plugging in a component with desirable functional requirements.

Let us consider process mining discipline. Today, there is a set of freeware and commercial process mining tools. Examples of these tools include *ProM* [9], which is a widely used research workbench containing more than 600 plug-ins. One of the most significant limitation of ProM is a inability to make an experiment workflow containing several routines from different plug-ins. *DPMine/P* framework and language [10] was intended as a remedy to fill up this gap and implemented as a set of ProM plug-ins using Java. Thus, *DPMine/C* implements DPMine workflow language as an independent C++ library [11].

One of the main VTMine/C purposes is to be a host application for DPMine/C library. The latter is integrated with VTMine/C as a separate plug-in that allows creating DPMine workflows through DPMine/C library functions. DPMine/C tools can edit and execute workflows composed of blocks. In its turn, the block type system is customizable by plug-ins.

In this paper we propose a concept of yet another modular framework. The paper discusses some aspects of the framework under development. We use C++ for the core implementation and Qt 4.8 library (cross-platform UI and software application development framework) for building a rich client application. The crossplatform programming language C++ was chosen for its speed. Furthermore, it allows developers to manage memory allocation. It is a crucial argument when processing large amounts of information, which often occurs

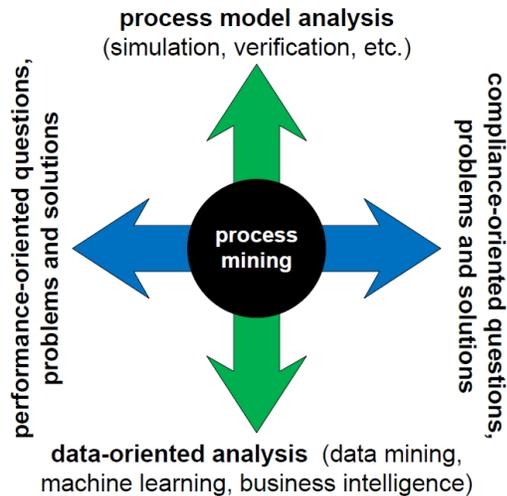


Figure 1. Process mining discipline emerged at a junction of process model analysis and data analysis.

in complex modelling tasks.

The rest of this paper is organized as follows. Section II describes the modular concept and key components of the tool. Section III discusses the lifecycle of a plug-in in VTMine/C. An example of interaction between the core platform and an existing plug-in in a form implemented by a Petri net Editor, is presented in Section IV. Finally, Section V concludes the paper and summarizes the work done.

II. FRAMEWORK STRUCTURE

VTMine/C target is to provide a fitting environment for additional plug-ins or components. From the users' point of view they interact with plug-ins through menus, toolbars, etc. From a technical standpoint they are represented as controls. The users work with a set of *resources*, using routines for processing them. The user can also manage resources through *projects* and *solutions*. The plug-ins register controls to give the user access to their functionality. The user interacts with plug-ins indirectly, but through a standard well-defined interface provided by the framework core. The interface contains necessary controls such as menus, tool and instrumental panels and general purpose control containers. The latter are used for representing complex components like a graph editor.

One need to design an abstraction layer through which a vast range of third party components are allowed to extend the framework core and even other plug-ins and customize their behaviour. For this purpose the framework allows plug-ins to interact with each other and with the framework modules (Fig. 2) through specific interfaces. Thus, each plug-in is independent of particular implementations of other components except those it is dependent on. Strong abstraction makes system very flexible.

Let us now consider main framework components closer.

A. Plugin Manager

Plugin Manager is a central module of the core platform. It is responsible for loading plug-ins. Each plug-in is located

in a special directory.

The loading of plug-ins is a two-step process. At the first step (see Fig. 3) a list of plug-ins compatible with the platform is created. There are components that implement the base interface (*IBasePlugin*; see sect. III). It is checked whether the plug-ins are compatible with the framework or not. In case there are different versions of the same feature, the one with the latest version is added to the list. As a result of the first step we get a list of candidates for further verification.

Before discussing the second step, let us introduce *dependencies* between plug-ins *A* and *B*. If *A* is dependent on plug-in *B* version *N*, it means that *A* cannot be loaded before loading of plug-in *B* version *N* has been performed. In the case of the circular dependencies between two plug-ins (*A* and *B*), no one of them are loaded. This is because it is impossible for Plugin Manager to determine which one should be loaded first.

A plug-in can also determine its previous versions that it is compatible with. For example, plug-in *C* version 78 can be compatible with versions from 60 to 77 with regard to a certain feature (versions of the plug-in can be given as a range). It helps to determine whether a version conflict exists or whether components dependent on some of the previous versions of a plug-in can be loaded (see Fig. 5).

At the second step the dependencies of the plug-ins are checked, components are loaded (Fig. 4) and registration of components is carried out. The plug-ins register all their interfaces, both visual and non-visual. This procedure is discussed in Section III in detail.

B. Resources

Let us consider resources. A resource is a typed envelop for data being processed and transformed by VTMine/C plug-ins. The type of resource is registered in *Resource Type Manager* by a plug-in that introduces such kind of resources for the framework (Fig. 6). For example, there is a plug-in registering a "Petri net" resource type. Often, the same plug-in registers not only new resource types, but also additional components for working with the resources of that type. For example above, the plug-in can also register an editor for Petri nets.

Some kinds of resources have a file representation and one or more filename extensions in order for the framework to be able to determine *resource types* of the imported files. The resources are objects that can be created, copied, deleted either by the user or any other framework component. For example, resources can be produced and processed by plug-ins. Depending on its functional purpose resources can be rather simple (like regular numbers and strings) as well as complex structures (various models, logs, etc.).

Resources are managed individually by special types of components, not by the framework. All resources inherit from a Basic Resource class. It declares common attributes of resources. There are no limits for the number of resource types. One can develop a new resource type by inheriting an existing resource. Resources are determined by resource types which do not contain data itself, but rather declare their attributes.

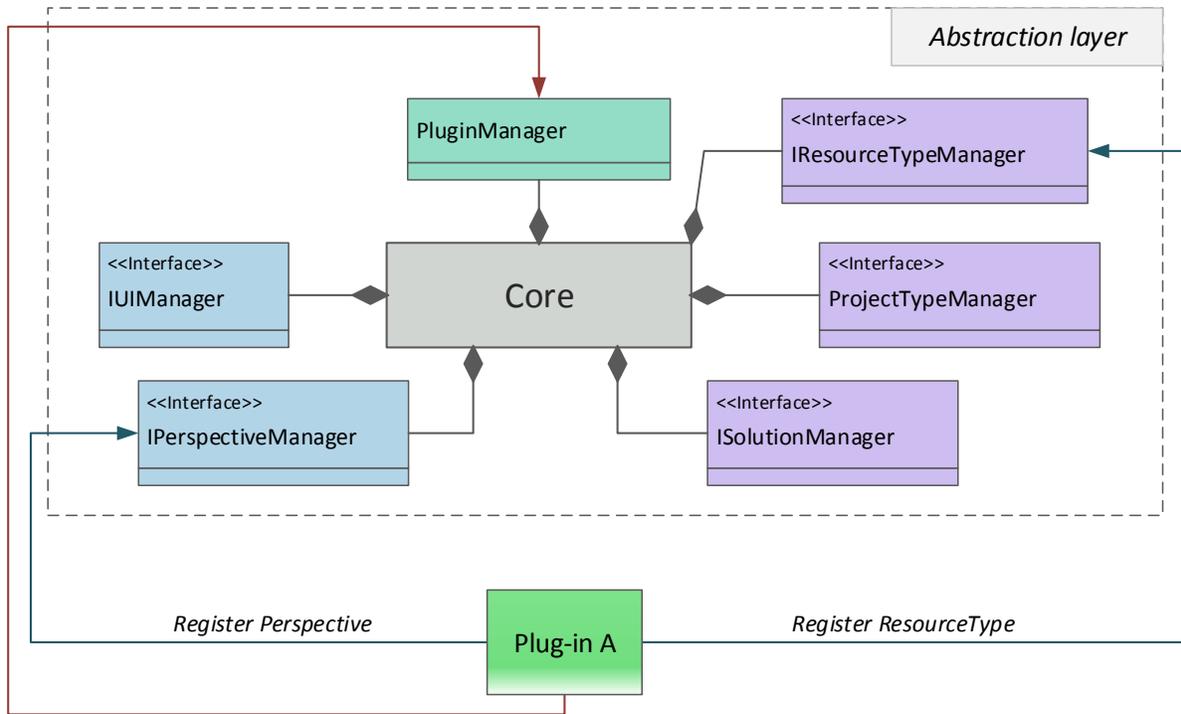


Figure 2. Cooperation of plug-in *A* and VTMine/C framework modules

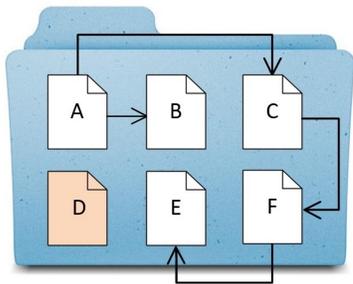


Figure 3. Example of directory of possible VTMine/C plug-ins. Arrows indicate the dependencies between the plug-ins. Plug-in *D* does not implement `IBasePlugin` so it is not going to be loaded

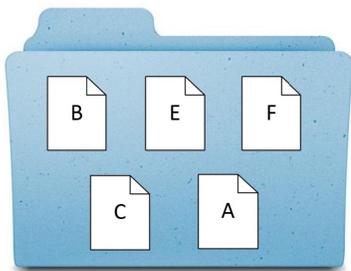


Figure 4. Example of plug-ins list after performing the loading routine. The order is important

Each resource can be associated with some so-called *resource doer* objects. A resource doer is a special object which can perform some operations with given resources. Resource doers are exported to the core platform by plug-ins. For example, some resource doer for a Petri net can open a Petri net editor. While another resource doer can perform some verification procedure with the Petri net. From the users' perspective they always work with resource doer objects. Access to the resource doer functionality is performed through controls. The plug-ins register controls during the loading process.

C. Projects and Solution Manager

Projects are used for resources' interaction. Some resource files can be grouped into a project. The project represents these resources as a tree-based structure including so-called folders as a tool for grouping related resources.

A *project type* describes projects the system can work with. The project type describes whether resources can be used in this very project (Fig. 6). Project types can differ in characteristics, for example, resources included by default. A project can also be stored as an XML-based file which describes properties of the project.

A *solution* is a high-level tool for organizing some individual projects. From the file system perspective projects are subdirectories of the solution's folder. *Solution Manager* controls how projects are stored.

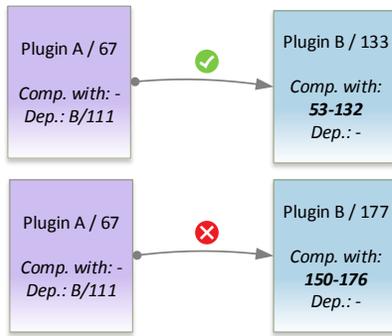


Figure 5. Example of transitive dependency between plug-ins. In the first case *A* can be loaded, in the second it cannot

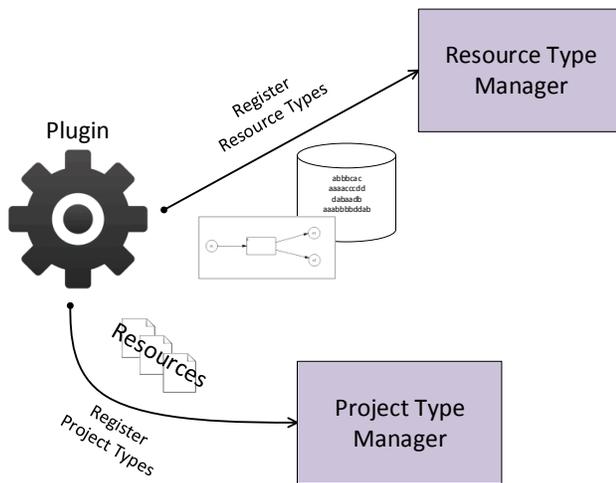


Figure 6. Resource Types and Resource Projects are registered by plug-ins

D. Perspectives

A *perspective* is a named customized set of visual controls including menus, toolbars, tool panels, and containers for custom controls (Fig. 7). A perspective aggregates visual controls with a view to providing users with necessary tools or controls for managing some kinds of resources or to perform specific tasks.

Switching of perspectives leads to a change in the number of menus, toolbars and panels. A *Main Perspective* is always enabled in the core platform. It stores a minimal set of controls so that the program remains manageable. It contains, for example, such controls as a menu which is responsible for switching between perspectives or an exit menu. Custom perspectives are created by VTMine plug-ins.

III. PLUG-IN LIFECYCLE

A plug-in is a component that adds new features to the framework or extends functionality of other plug-ins. As a

component-based application the tool provides an ability to interact with third-party components developed for VTMine/C. They appear in the form of shared libraries, dynamic link libraries (.dll) in Windows and dynamically linked shared objects libraries (.so) in Linux.

In VTMine/C all plug-ins contain a class implementing `IBasePlugin` interface. The interface does not declare any component functions. There is a special method that gives minimal required information about a plug-in to the framework. This information is enough to load the plug-in. It is represented in the form of *Descriptor*. A *Descriptor* is a structure containing the following information:

- Plug-in name
- Plug-in developer
- Plug-in version
- Plug-in dependencies
- Some additional information

Let us consider other methods of `IBasePlugin` interface. There is a method called `registerMyself` (it is responsible for the plug-in registration procedure and is executed immediately after loading a component) and a method handling *commands*. These functions are discussed comprehensively below.

The plug-ins depending on their function can perform the following tasks:

- Expand framework GUI by introducing new controls
- Represent doer objects (viewer or editor)
- Be a special kind of doers in the form of various implementing algorithms that perform some actions
- Register new resources

Let us consider the lifecycle of a plug-in.

A. Loading

As it was written above, a plug-in is loaded if it satisfies a required interface and all components associated with it have been already loaded.

Next, the plug-in is expected to prepare a platform for interaction.

B. Registration

When the plug-in is loaded, it can register itself in the core platform. VTMine/C provides the plug-ins with well-defined interfaces of the managers discussed in Section II. The plug-in consumes their features and enable to add custom features. The plug-in registers in the framework various components like resource types or *menu items*. It is important to point out that, in fact, the plug-ins does not access GUI itself. The plug-in can only register controls regardless of their graphical interfaces. If necessary, the platform wraps them into graphical controls.

At this stage of development all components, plugging in to VTMine/C, have the same access rights and are able to change any module or module properties in the core platform. This approach is flexible but insecure. For security purposes the platform checks whether the components support required

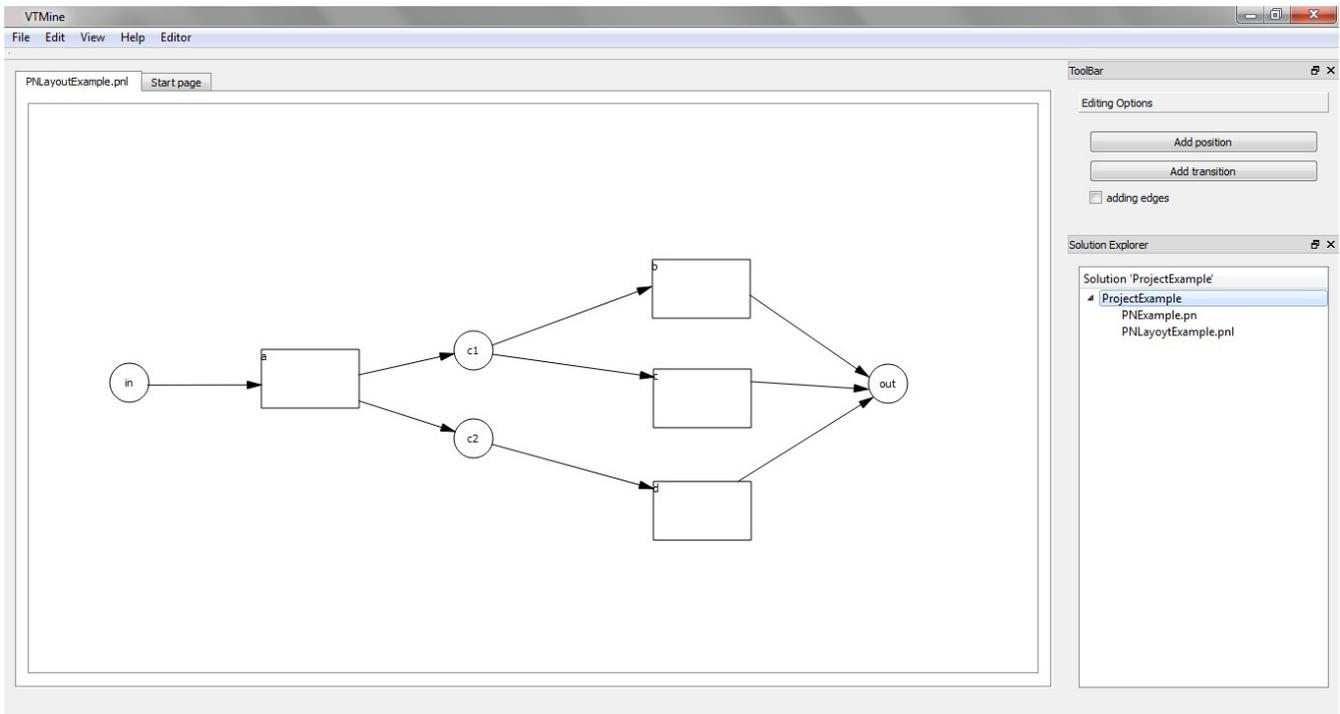


Figure 7. Example of VTMine main window where a Petri net editor plug-in has been registered

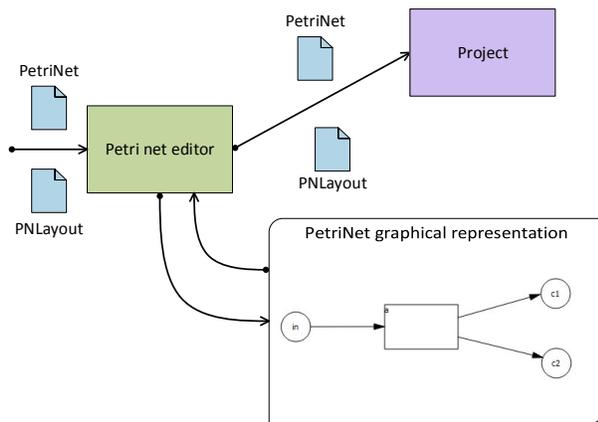


Figure 8. Scheme of the work of a Petri net editor plug-in

interfaces and depending on the obtained results provides plug-ins with different access rights to the framework modules.

C. Further Maintenance

When all components are loaded and registered the framework is ready for interaction. Further execution of the plug-in depends on the user. Each event issued from the user is a *command*. A command has a unique description which includes information about its producer and consumer. As an event occurs, the framework handles the command and determines the consumer. The platform just transfers the command to the plug-in without being aware of the command's function. Next, the consumer invokes a required tasks operation.

The next section illustrates an example of interaction between a Petri net Editor plug-in and the framework.

IV. PLUG-IN EXAMPLE

Let us look at a plug-in exporting Petri net editor to the framework. First, let us see how the plug-in looks like and what its main attributes are.

The plug-in is aimed to 1) visualize Petri net models, 2) edit them and 3) export Petri nets given as VTMine resources to files. In editing mode basic Petri net items such as positions, transitions and edges can be added using a toolbar that is also registered by the plug-in. The plug-in registers a special widget, placed in a container for custom controls discussed above, that is responsible for Petri net graphical representation. The Petri net editor plug-in that has been registered in the VTMine framework is shown in Fig. 7.

A. Lifecycle of a component

It can be mentioned, that the Petri net editor has no dependencies on other plug-ins, so it can be freely loaded by Plugin Manager during the loading procedure.

The editor uses different framework managers to register its components. Every plug-in “knows” the framework and has access to its modules. Let us look at the process of plug-in registration.

- 1) The Petri net editor plug-in registers its menus in Menu Manager. The manager does not create any graphics representation, but adds a description of menu items. Menu Manager stores a list of menus. So the plug-in

adds a new menu containing menu identifier, title and items.

- 2) The plug-in registers a toolbar using Toolbar Manager. The process of toolbar registration is similar to the process of menu registration.
- 3) UI Manager obtains a graphical representation of controls corresponding to their descriptions which have been registered at the previous step. Moreover, the editor gets the central widget of the main form from UI Manager to place a widget that is responsible for the Petri net's graphical representation.
- 4) The plug-in registers a new perspective using Perspective Manager. Furthermore, the editor binds its menu and toolbar to the perspective.
- 5) The Petri net editor uses two special kinds of resources. These are PetriNet and PMLayout resources. The first keeps the internal representation of a Petri net as an object model. The second keeps additional information about the Petri net's graphics primitives as they are represented on the screen. The plug-in tries to register these resource types, if they have not been already registered.

B. Plug-in Work

The editor can both import PetriNet alone or be coupled with PMLayout and give the ability to edit graphical representation of the Petri net. Moreover, the plug-in can export current PetriNet or PMLayout as a new VTMine resource.

When the framework uses the editor, it creates a new Project in Project Type Manager. The project manages the resources that are imported and exported during the editor's lifetime. A work scheme of the plug-in is shown in Fig. 8.

V. CONCLUSION

In this paper the concept VTMine/C framework was discussed. The core of the framework is being changed continuously, because the project implementation is in its initial stage. Some of the components have already been done whereas others are still need to be developed and implemented. We expect some concepts drift concerning UI improvement.

Now the main work is to perform integration between VTMine/C and DPMine/C. Moreover, we consider to continue developing under some public license to involve the community interested in this work. The web site of VTMine project is <https://prj.xiart.ru/projects/vtmine>.

ACKNOWLEDGMENT

The study was implemented in the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE).

REFERENCES

- [1] D. Bose. (2010, November) Component Based Development. Cornell University Library. [Online]. Available: <http://arxiv.org/ftp/arxiv/papers/1011/1011.2163.pdf>
- [2] W. M. P. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
- [3] K. Moir. (2012) The Architecture of Open Source Applications - Eclipse. [Online]. Available: <http://www.aosabook.org/en/eclipse.html>
- [4] How to Extend RapidMiner 5. [Online]. Available: <http://rapidminer.com/wp-content/uploads/2013/10/How-to-Extend-RapidMiner-5.pdf>
- [5] Miranda IM website. [Online]. Available: <http://wiki.miranda-im.org/>
- [6] A. Campos, B. Lane, N. Clark, S. Jassal, and S. Hitchner. (2007, June 2) Conceptual Architecture of Firefox. [Online]. Available: <http://web.uvic.ca/~hitchner/assign1.pdf>
- [7] Hudson website. [Online]. Available: <http://www.eclipse.org/hudson/>
- [8] Notepad++ website. [Online]. Available: <http://notepad-plus-plus.org/>
- [9] H. Verbeek, J. Buijs, B. Dongen, and W. Aalst, "ProM 6: The Process Mining Toolkit," in *Proc. of BPM Demonstration Track 2010*, ser. CEUR Workshop Proceedings, M. L. Rosa, Ed., vol. 615, 2010, pp. 34–39.
- [10] S. Shershakov, "DPMine/P: modeling and process mining language and ProM plug-ins," in *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, A. N. Terekhov and M. Tsepkov, Eds. ACM New York, NY, USA, 2013. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2556622&CFID=415147702&CFTOKEN=35395117>
- [11] —, "DPMine/C: C++ library and graphical frontend for DPMine workflow language," in *Proceedings of the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering, SYRCoSE 2014*, 2014, in press.

Extended Finite State Machine based Test Derivation Strategies for Telecommunication Protocols

Natalia Kushik
Tomsk State University
Tomsk, Russia
TELECOM SudParis
Evry, France
Email: ngkushik@gmail.com

Anton Kolomeez
Tomsk State University
Tomsk, Russia
Email: anton.kolomeez@gmail.com

Ana R. Cavalli
TELECOM SudParis
Evry, France
Email: ana.cavalli@it-sudparis.eu

Nina Yevtushenko
Tomsk State University
Tomsk, Russia
Email: ninayevtushenko@yahoo.com

Abstract—In this paper, we consider the problem of test derivation based on an Extended Finite State Machine (EFSM) that is widely used for describing the behavior of telecommunication protocols and software. An EFSM augments a classical Finite State Machine (FSM) with context variables, input/output parameters and predicates. Tests based on various coverage criteria for EFSMs do not capture many functional faults and thus, there is a strong need for tests checking functional properties. Moreover, since there are no constructive necessary and sufficient conditions for checking whether two arbitrary EFSMs are equivalent, most methods are based on some kind of a transition tour, despite of the fact that such methods do not provide test suites with the guaranteed fault coverage. Given possibly nondeterministic and partial EFSM, we consider a transition tour of an FSM obtained by the simulation of the initial EFSM and provide some experimental results that such a test suite detects a number of inconsistencies in available protocol implementations with respect to protocol specifications. Since a transition tour augmented with state identifiers is known to have the higher fault coverage, we also discuss how state identifiers can be generated without facing the state explosion problem. Correspondingly, we consider FSM slices that are obtained by deleting from the initial EFSM all the context variables and possibly, input and output parameters. As the obtained FSM can be nondeterministic, a state identifier should contain separating sequences for pairs of states and we adapt the known techniques for deriving separating sequences for nonobservable partial FSMs.

I. INTRODUCTION

In this paper, we consider the test derivation based on the model of an Extended Finite State Machine (EFSM) that is widely used for describing protocols and software, see, for example, [1], [2]. We underline that tests based on various EFSM coverage criteria do not capture many functional faults [3], [4] and thus, there is a strong need for tests derived against formal behavioral models.

An EFSM augments a classical Finite State Machine (FSM) [5] with context variables and input and output parameters.

Since there are no constructive necessary and sufficient conditions for checking whether two arbitrary EFSMs are equivalent (as for instance, the bisimulation for classical FSMs), most test derivation strategies are based on some kind of a transition tour, despite of the fact that such methods do not provide test suites with the guaranteed fault coverage. Moreover, differently from classical FSMs, there is the well known execution problem for tests derived against EFSMs, according to the necessity for providing appropriate values for internal context variables. In order to avoid this problem, many methods for deriving functional tests use an appropriate EFSM slice with the FSM behavior. In this paper, we consider three different EFSM slices. The first one is an FSM that is derived based on the simulation of a given EFSM [6], [7]. However, in this case, we meet the well known state explosion problem and the FSM is usually generated up to the given number of states or up to the given length of input sequences. A transition tour is then derived for the obtained FSM and we illustrate that such test detects a number of inconsistencies in available protocol implementations [8], [9].

The quality of a transition tour can be improved using distinguishing sequences [2] for final states/configurations of traversed transitions. Moreover, in order to minimize a test suite, each distinguishing sequence should distinguish as much states/configurations as possible [10]. One way to derive such distinguishing sequences is to use a corresponding distinguishing EFSM [2] that in fact, is a product of initial EFSMs with corresponding initial states. However, this approach is well elaborated only for two configurations and there still is the execution problem for an obtained distinguishing sequence, since not every parameterized input sequence that takes the product of two initial EFSMs from the initial state to a *fail* state is executable. Another way is to distinguish not configurations but states of the EFSM using context-free slices of an EFSM that are very close to classical FSMs and correspondingly, there is no problem to execute a derived distinguishing sequence [11]. In this paper, we consider two such slices. However, in this case, a corresponding FSM can be nondeterministic, i.e., distinguishing sequences become separating sequences.

The work is partially supported by RFBR grant No. 14-08-31640 mol_a.

An input sequence is a separating sequence for a given set of FSM states if for each two different states of the set, the sets of output responses of the FSM at these states to the input sequence do not intersect. As an obtained FSM can be partial and nonobservable, in this paper, methods proposed in [12], [13] for deriving separating sequences are adapted to partial and nonobservable FSMs.

Correspondingly, the main contribution of the paper is a method for deriving a separating sequence for a set of k states, $k > 1$, of a nondeterministic FSM that can be partial and nonobservable. Experimental results are involved when talking about another contribution. These results clearly show that the EFSM slices fit very well for deriving high quality tests and thus, test derivation strategies could be further improved based on various EFSM slices. One of directions for future work includes slicing based on time variables when time constraints are involved in the EFSM description [14].

The rest of the paper is organized as follows. Section 2 contains some preliminaries for EFSMs, while Section 3 describes how test suites are generated based on three FSM slices of the initial EFSM. Experimental results on the transition tour quality are reported for some telecommunication protocols and methods for deriving distinguishing/separating sequences for EFSM states based on EFSM slices are discussed. Section 4 concludes the paper.

II. PRELIMINARIES

EFSM Model

A *finite state machine (FSM)*, or simply a *machine* throughout this paper is a 5-tuple $\mathbf{S} = \langle S, I, O, h_S, S' \rangle$, where S is a finite nonempty set of states with a nonempty subset S' of initial states; I and O are finite input and output alphabets; and $h_S \subseteq S \times I \times O \times S$ is a *behavior (transition) relation*. If $|S'| = 1$ then the machine is *initialized*, otherwise it is *non-initialized (weakly initialized)*. An FSM is *nondeterministic (NFSM)*, if for some pair $(s, i) \in S \times I$ there exist several pairs $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$, otherwise \mathbf{S} is *deterministic*. If for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$ then the FSM is *complete*, otherwise it is *partial*. If for each triple $(s, i, o) \in S \times I \times O$ there exists at most one state $s' \in S$ such that $(s, i, o, s') \in h_S$ then the FSM is *observable*, otherwise it is *nonobservable*. In usual way, the FSM behavior is extended to input sequences.

Given an FSM $\mathbf{S} = \langle S, I, O, h_S, S' \rangle$, two states $s_1, s_2 \in S$ are *compatible* if for each input sequence $\alpha \in I^*$ the sets of output responses at these states to α coincide, i.e. $out(s_1, \alpha) = out(s_2, \alpha)$. Two states are *distinguishable* if there exists an input sequence α such that α is a defined input sequence at both states s_1 and s_2 and $out(s_1, \alpha) \neq out(s_2, \alpha)$. The FSM \mathbf{S} is *reduced* if its states are pair-wise distinguishable. States s_1, s_2 of \mathbf{S} are *separable* if there exists an input sequence $\alpha \in I^*$ such that $out(s_1, \alpha) \cap out(s_2, \alpha) = \emptyset$; in this case, α is a *separating* sequence of states s_1 and s_2 . If there exists an input sequence α that separates every two distinct states of the set S' , then α is a *separating* sequence for the set S' .

An extended finite state machine (EFSM) [2], [6] A is a pair (S, T) of a set S of states and a set T of transitions between states, such that each transition $t \in T$ is a tuple

$(s, i, o, P, v_p, o_p, s')$, where $s, s' \in S$ are the starting and final states of a transition; $i \in I$ is an input with the set D_{inp-i} of possible vectors of corresponding input parameter values, $o \in O$ is an output with the set D_{out-o} of possible vectors of output parameter values; P, v_p , and o_p are functions, defined over input parameters and context variables. By definition, $P : D_{inp-i} \times D_V \rightarrow \{True, False\}$ is a predicate where D_V is the set of context vectors; $o_p : D_{inp-i} \times D_V \rightarrow D_{out-o}$ is an *output parameter update* function; $v_p : D_{inp-i} \times D_V \rightarrow D_V$ is a *context update* function.

According to [2], we use the following definitions. Given an input i and a vector $\mathbf{p} \in D_{inp-i}$, the pair (i, \mathbf{p}) is called a *parameterized input*; if there are no parameters for the input i then i is a *non-parameterized* input. A sequence of parameterized inputs (possibly some of them are non-parameterized) is called a *parameterized input sequence*. A context vector $\mathbf{v} \in D_V$ is called a *context* of A . A *configuration* of A is a pair (s, \mathbf{v}) . Usually, the initial state and the initial configuration of the EFSM are given; thus, given a parameterized input sequence of the EFSM, we can calculate a corresponding parameterized output sequence by simulating the behavior of the EFSM under the input sequence starting from the initial configuration.

An EFSM is *consistent* if for each transition at state s with input i , every element in $D_{inp-i} \times D_V$ evaluates exactly one predicate to true among all predicates guarding the different transitions with the starting state s and input i ; in other words, the predicates are mutually exclusive and their disjunction evaluates to true. An EFSM A is *completely specified* if for each pair $(s, i) \in S \times I$, there exists at least one transition at state s with the input i . The authors of most papers develop test derivation strategies for consistent and completely specified EFSMs. However, such EFSMs are rarely met when building protocol specifications at high abstraction levels.

The equivalence and distinguishability relations for EFSM configurations are defined similar to those over FSM states. Two initialized EFSMs are *compatible* if their initial configurations are compatible. Differently from FSMs, we still lack necessary and sufficient conditions for establishing whether even two complete and consistent EFSMs are equivalent. Two states of an EFSM are *separable* if there exists a (parameterized) input sequence such that at these states the sets of parameterized output responses of the EFSM to this input sequence do not intersect (for any values of context variables). In other words, if two states s and s' of the EFSM are separable then each two configurations at these states are separable.

When the specification domain of each context variable and of each input parameter is finite an EFSM A can be unfolded to an equivalent FSM, written $FSM_{sim}(A)$, by simulating its behavior with respect to all possible values of context variables and input vectors. The equivalence means that the set of traces of the FSM coincides with the set of parameterized traces of the EFSM. Given a state s of EFSM A , a context vector \mathbf{v} , an input i and the vector \mathbf{p} of input parameters, we derive the transition from configuration (s, \mathbf{v}) under input (i, \mathbf{p}) in the corresponding FSM. We first determine the outgoing transition $(s, i, o, P, v_p, o_p, s')$ from state s where the predicate P is true for the input vector \mathbf{p} and the context vector \mathbf{v} , update the context vector to the vector \mathbf{v}' according to the assignment v_p of this transition, determine the parameterized

output (o, \mathbf{w}) and add the transition $((s, \mathbf{v}), (i, \mathbf{p}), (o, \mathbf{w}), (s', \mathbf{v}'))$ to the set of transitions of the FSM $FSM_{sim}(A)$. The number of states of the obtained FSM equals the number of different configurations (s, \mathbf{v}) of the EFSM that are reachable from the initial configuration. If an EFSM is consistent and completely specified, the corresponding FSM is complete and deterministic. Two EFSMs are equivalent if and only if their corresponding FSMs are equivalent [7]. When the specification domain of some context variable and/or some input parameter is infinite or the number of generated transitions becomes huge, the EFSM behavior is simulated up to the given number of transitions or up to the given length of input sequences.

III. DERIVING TEST SUITES FOR DETECTING FUNCTIONAL FAULTS

A. Transition tour of the slice FSM_{sim}

When a test suite is derived using FSM based methods, the high quality of the test suite is guaranteed by traversing each transition of an FSM under test and by distinguishing the final state of a traversed transition from other states. However, almost all FSM based methods are developed for complete deterministic FSMs, while the FSM $FSM_{sim}(A)$ is usually partial and nondeterministic. Moreover, as discussed above, sometimes only a part of this FSM can be derived due to the well known transition explosion problem. In order to avoid this problem the maximal number of states of the $FSM_{sim}(A)$ is limited by some integer B or the length of input sequences used for the simulation is limited by some integer l . In the former case, all the states corresponding to configurations (s, \mathbf{v}) with the numbers that are greater than B are marked by a special state *DNC* (*DON'T CARE* state) where the self-loops labeled with all input/output pairs can be added. Two ways are then appropriate when deriving a test suite for the obtained $FSM_{sim}(A)$.

- 1) Transitions with the DNC state are deleted from the $FSM_{sim}(A)$ and a test suite is derived for a partial FSM [15].
- 2) A test suite is derived for a completely specified FSM $FSM_{sim}(A)$ and then the test suite is 'refined' by deleting all suffixes of test sequences that lead to the DNC state.

When performing experiments with telecommunication protocols, we tried the first approach and derived a transition tour that is known to detect all output faults at all traversed transition. It is also known that such a test suite does not detect all transfer, predicate and assignment faults and in order to report the quality of such test suites we use experimental results with available implementations of some telecommunication protocols.

The protocol **IRC** [8], [16] we have experimented with is used for organizing the real time message exchange between internet nodes. The EFSM specification is partial and non-deterministic according to several reply options to the same query.

The behavior of the FSM $FSM_{sim}(A)$ is included into the behavior of the initial EFSM that is derived using the RFC specification. For this protocol, an FSM that covers all configurations cannot be derived, since specification domains

of the context variables are infinite. When completing the connection a client sends the message *QUIT*, i.e., the message *QUIT* takes any IRC implementation to the initial state. The EFSM A that has been extracted from the RFC specification [16] is an initialized EFSM that has four states, 47 transitions, 12 inputs, 25 outputs, 6 context variables and 17 input and output parameters. After limiting the number of states by $B = 9$ and deriving the FSM $FSM_{sim}(IRC)$ with 34 inputs, a test suite has been derived as a transition tour of the obtained FSM [8]. This test contains 265 input sequences with the total length of 1164 inputs counting *QUIT* input as the reset. The test was downloaded into the data base of the software *Tester* [17] for testing a free available implementation *ngIRCd* (version 16) that is widely used as a server IRC implementation. The software *ngIRCd* was downloaded from the developer web site and was compiled by A. Shabaldin using the utility *strict - rfc*. Three inconsistencies have been detected by the test. First, there was a wrong reply code to the *NICK* command with the empty parameter. Another inconsistency occurred due to the incorrect server use of the *Nickname* that is already occupied, while the third inconsistency was related to the wrong reply to the *MODE* command that is used without the *Nickname* but with the parameter for setting a communicating mode.

TFTP [9] is a simple file transfer protocol that is used for reading and writing files from/to a remote server. It is generally used to move files between machines of different networks implementing User Datagram protocol and the simplicity of the protocol makes it very popular. Following the RFC specification [18] a special EFSM with four states, 11 transitions and a single context variable that represents a timer, i.e., a clock variable [9], has been derived. Since a context variable is a clock variable, a method presented in [19] has been used for simulating such extended machine in order to obtain an equivalent FSM where for the sake of simplicity, only the part that is responsible for getting files from the server has been modelled. A test suite was derived as a transition tour of the obtained FSM. Experiments with two implementations supporting TFTP, namely, class *TFTPServer* defined in the *commons - net - 2.0.0* library developed by Apache and *atftpd* Linux server developed by Jean-Pierre Lefebvre are reported in [9]. Some mismatching has been detected between these implementations and the protocol specification. In the *TFTPServer* an acknowledgement with the unset packet number has been ignored while the *atftpd* implementation replies to acknowledgements incrementing their numbers that does not match the protocol specification.

POP3 [20] is the Post Office Protocol of the third version that is used at the application-layer by local e-mail clients to retrieve messages from the server. Different webmail service providers like *Gmail* or *Yahoo* support this protocol and thus, corresponding implementations should be thoroughly tested. Following the RFC1939 specification an EFSM E describing the behavior of the POP3 protocol [21] with four states and two context variables has been derived. The EFSM was then unfolded to an equivalent FSM with six states and 106 transitions [22]. Similar to previous cases, a test suite has been derived as a transition tour. The test suite has detected an inconsistency in the POP3 implementation *tpop3d - 1.5.5* that is related to the incorrect processing of the double use of *DELE* command for the same message.

Some experiments were performed with other protocols described as EFSMs where a test suite has been derived as a transition tour of a corresponding unfolded FSM. Experimental results show that the fault coverage for such test suites is around 100% for output faults and approximately 60% for other faults such as transfer, predicate and/or assignment faults as they are defined in [6]. In order to enhance the fault coverage the authors of different papers (see, for example [2]) propose to add distinguishing sequences for the final configurations of each traversed transition and according to the results on FSM based test derivation [10], in order to minimize the length of a resulting test suite, each distinguishing sequence should distinguish as much states as possible. However, in order to escape the state explosion and the execution problems we propose to distinguish not configurations but states of the initial EFSM using corresponding context-free slices. Since such slices usually have the nondeterministic behavior, distinguishing sequences become separating sequences and as obtained nondeterministic slices can be partial and nonobservable, the existing methods for deriving separating sequences [13], [12] have to be adapted to this class of nondeterministic FSMs.

B. Context-free EFSM slice

Here we consider a slice $Slice_{context-free}(A)$ of an EFSM A that does not have context variables. We follow the approach in [11], but a proposed technique allows to derive such a slice preserving more transitions of the initial EFSM [3]. The idea behind the approach is to delete transitions from the initial EFSM which have predicates that significantly depend on values of context variables. However, some of such transitions can be preserved using the following property. For example, if P is the disjunction of predicates P_1 and P_2 , and P_1 does not significantly depend on the values of context variables then a transition with the predicate P will be fired for appropriate values of input parameters where P_1 is true, i.e., a transition with the predicate P can be replaced by the same transition with the predicate P_1 . In general case, such replacing is valid if the predicate P can be represented as a function of P_1 and P_2 , $P = f(P_1, P_2)$, where P_1 does not significantly depend on context variables, and $f(1, 0) = f(1, 1) = 1$. At the next step, all the context variables and functions for updating these variables are deleted from the obtained EFSM.

As an example, consider $P = a_1a_2 \vee v_1v_2$, $P_1 = a_1a_2$, $P_2 = v_1v_2$, where a_1 and a_2 are Boolean input parameters while v_1 and v_2 are Boolean context variables. In this case, in the FSM slice a transition $(s, i, o, P, v_p, o_p, s')$ can be replaced by a transition (s, i, o, P_1, s') and correspondingly, only input (external) parameters have to be set for traversing the transition.

By construction, the $Slice_{context-free}(A)$ has no context variables, i.e., has an FSM behavior. Nevertheless, this slice has input parameters, i.e., parameterized inputs should be considered when deriving a test suite. Correspondingly, using a context-free slice two configurations (s, \mathbf{v}) and (s', \mathbf{v}') of the initial EFSM can be distinguished using FSM based methods if states s and s' are distinguishable in the $Slice_{context-free}(A)$. The $Slice_{context-free}(A)$ can have predicates which depend on input parameters and this should be taken into account when deriving a distinguishing sequence. As usual, for deriving a

distinguishing sequence for two states we consider a corresponding successor tree (or a product) [5] but this construction is augmented with checking conditions for predicate satisfiability and determining a corresponding satisfying assignment [3]. To the best of our knowledge, there is no general method how to solve the problem for an arbitrary predicate but for most protocols such predicates are described using Boolean functions or systems of linear comparisons over integers or rational numbers. If all the predicates are Boolean functions then the satisfiability problem is reduced to the well known SAT problem and there are efficient algorithms for its solving, see, for example [23], [24]. If predicates are represented as linear expressions then there are methods how to solve a corresponding system of linear comparisons [25]. According to performed experiments with some protocols [3], a test suite augmented with distinguishing sequences derived using $Slice_{context-free}(A)$ additionally detects a number of single and double transfer, predicate and assignment faults in protocol implementations.

C. Using separating sequences of the underlying FSM slice

Similar to the context-free slice, another FSM slice of the initial EFSM can be derived. Given an EFSM A , we derive an FSM $FSM(A)$ by deleting all context variables, input and output parameters, predicates, and update functions, i.e., each transition becomes a classical FSM transition containing starting and final states and an input/output pair i/o . By construction, the $FSM(A)$ can be nondeterministic, partial and nonobservable. Similar to the previous cases, a test suite can be derived based on a transition tour of $FSM_{sim}(A)$ (Section 3.1) augmented with separating sequences for each pair of different states of the $FSM(A)$ for which such a separating sequence exists. In order to minimize a test suite separating sequences which distinguish subsets of states of $FSM(A)$ are used. For this purpose, we adapt the algorithm proposed in [12] for separating two complete observable initialized FSMs for separating a subset S' of states of a possibly nonobservable FSM. We first propose a corresponding procedure for complete nonobservable machines and then discuss how it can be used when deriving separating sequences for partial nonobservable FSMs.

When deriving a separating sequence for FSMs we are interested in pairs of FSM states. A *pair* of states is an unordered state pattern of length two denoted as $\overline{s_p, s_q}$ with $s_p, s_q \in S$; if $s_p = s_q$ then the pair is a *singleton* $\overline{s_p, s_p}$. Given an input/output pair i/o and a state s_p , the set $next_state(s_p) = \{s \in S | (s_p, i, o, s) \in h_S\}$ is called an *i/o-successor* of state s_p . Given an input/output pair i/o and a pair $\overline{s_p, s_q}$, the *i/o-successor* of $\overline{s_p, s_q}$ is the set of different pairs of *i/o*-successors of states s_p and s_q (if such successors exist for both states s_p and s_q). In other words, the pair $\overline{s'_p, s'_q}$ belongs to the *i/o*-successor of the pair $\overline{s_p, s_q}$ if $(s_p, i, o, s'_p) \in h_S$ and $(s_q, i, o, s'_q) \in h_S$. The *i/o*-successor of the pair can contain a singleton $\overline{s_k, s_k}$ if s_k is included into the *i/o*-successor of both states s_p and s_q . Given an input i , the *i-successor* of $\overline{s_p, s_q}$ is the union of the *i/o*-successors of $\overline{s_p, s_q}$ for all possible outputs $o \in O$. The *i*-successor is empty if for each $o \in O$ the pair $\overline{s_p, s_q}$ has no *i/o*-successor.

Procedure 1 for deriving a shortest separating sequence for a subset S' , $|S'| \geq 2$, of a possibly nonobservable FSM

Input: FSM S that can be nonobservable and a subset $S' \subseteq S$, $|S'| \geq 2$

Output: A shortest separating sequence for S or the message "There is no separating sequence for the subset S' "

Derive a truncated successor tree for the FSM S . The root of the tree is labeled with the set of the pairs $\overline{s_p, s_q}$, $s_p, s_q \in S'$, $p < q$; the nodes of the tree are labeled by sets of pairs of the set S . Edges of the tree are labeled by inputs and there exists an edge labeled by i from a node P at level j , $j \geq 0$, to a node Q if Q is the union of the i -successors over all pairs of P . The set Q contains a singleton if i/o -successors of some pair of P coincide for some $o \in O$. If the union of i -successors of pairs from P is empty then the set Q is empty.

Given a node P at the level k , $k > 0$, the node is *terminal* if one of the following conditions holds.

Rule-1: P is the empty set.

Rule-2: P contains a set R that labels a node at a level j , $j < k$.

Rule-3: P contains a singleton.

If the successor tree has no nodes labeled with the empty set, i.e., is not truncated using Rule-1 then Return the message "There is no separating sequence for a subset S' ". Otherwise,

Determine a path with minimal length to a node labeled with the empty set;

Return separating sequence as the input sequence α that labels the selected path.

End

Theorem 1. *Given a subset S' of states of an FSM S , the set S' has a separating sequence if and only if a truncated successor tree returned by Procedure 1 contains a node labeled by the empty set. Moreover, if all the branches of the tree are truncated by applying Rules 2 and/or 3 then a separating sequence for the set S' does not exist.*

Proof: Let an input sequence $\alpha = i_1 i_2 \dots i_n$ label a path to a node with a set $P \neq \emptyset$ of pairs of states. If $|\alpha| = n$ then α traverses non-terminal nodes labeled with the sets P_1, P_2, \dots, P_{n-1} . The set $P_0 = S'$ and the set $P_n = P$. By construction, the set P_{l+1} contains pairs of states for which $\exists o_l \in O$ such that all pairs from P_{l+1} are i_l/o_l -successors of pairs of P_l , $l \in \{1, \dots, n-1\}$. Therefore, a pair $\overline{s_p, s_q} \in P$ if and only if there exists an output sequence $\beta = o_1 o_2 \dots o_n$ such that s_p and s_q are α/β -successors of two different states of S' . Correspondingly, a singleton $\overline{s_p, s_p} \in P$ if and only if s_p is the α/β -successor of two different states in S' .

The set P has all pairs that are α -successors of two different initial states and α is a separating sequence when $P = \emptyset$. Thus, a sequence α that labels a path of the truncated successor tree is a separating sequence for the set S' if and only if this path is terminated by the node labeled by the empty set. On the other hand, by definition, a sequence α that labels a path to a node truncated by Rule-3 cannot be a prefix of a separating sequence of the set S' . Moreover, Rule-2 allows to truncate unpromising tree branches. In fact, let α be a separating sequence for S' that traverses a k -level node

labeled by a set P , and the successor tree has a j -level node labeled by a set R , such that $R \subseteq P$ and $j < k$. In this case, there exists a separating sequence for the set S' with the length less than $|\alpha|$. Thus, if there exists a separating sequence for the set S' then each shortest separating sequence labels a path in the truncated successor tree returned by Procedure 1. ■

Proposition 2. *Given a subset S' , $|S'| = m$, of states of the FSM S with n states, the length of a shortest separating sequence for S' is at most $2^{\binom{n}{2}} - 2^{\binom{n}{2} - \binom{m}{2}}$.*

Proof: Similar to [26], the length of a separating sequence for an FSM with n states and m initial states is bounded by the number of sets of state pairs that do not include pairs of initial states (Rule-2). The number of all sets of state pairs which are not singletons equals $2^{\binom{n}{2}}$ while the number of sets of state pairs including pairs of initial states equals $2^{\binom{n}{2} - \binom{m}{2}}$. ■

Despite of the fact, that the above upper bound is reachable [12], the performed experiments with randomly generated FSMs show that usually the length of a separating sequence for two states of an FSM (if such a sequence exists) is much shorter. Thus, the above approach might be useful when enhancing the fault coverage of an EFSM based test suite.

Deriving separating sequences for partial FSMs. Given an EFSM, the underlying FSM usually is not only nondeterministic and nonobservable but also partial. In order to adapt the above procedure to partial FSMs, the interpretation of the undefined transitions should be considered [27]. One of the widely used interpretations of an undefined transition is the augmentation of this transition by a loop labeled with a special output *IGNORE* or *NULL*. In this case, it is assumed that all undefined transitions will be augmented in the same way in any protocol implementation. In the second interpretation of an undefined transition, the transition is considered as a *DON'T_CARE* transition, i.e., this transition can be implemented as a transition to every state with every output. In this case, the transition can be implemented in an arbitrary way and at the first step of Procedure 1 (truncated tree derivation) given a pair $\overline{s_p, s_q}$ such that a transition under input i is defined only at state s_p as a transition to state s'_p , it is taken into account by adding all possible pairs $\overline{s'_p, s'_q}, s'_p, s'_q \in S$ as i/o -successors of the pair $\overline{s_p, s_q}$. However, it can happen that the input i cannot be applied at state s_q and correspondingly, the *DON'T_CARE* interpretation of this transition is not possible. For example, this can happen when considering a partial or a modular design when inputs of a given EFSM are outputs of another machine [28], [29]. In this case, the only solution is to consider inputs which are defined at each state of each pair of the set that labels a current node of the successor tree derived by Procedure 1.

As an example, consider an FSM represented in Fig. 1 where the set $S' = \{1, 2, 3\}$.

In the case, when undefined transitions are interpreted as forbidden actions, the FSM in Fig. 1 has no separating sequence. In fact, at state 3, a transition under input i_1 is not defined while at state 2 there are no transitions under input i_2 . On the other hand, when interpreting undefined transitions as loops with the *NULL* output Procedure 1 can be applied.

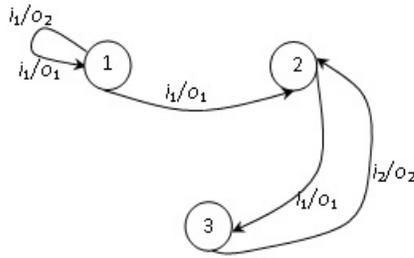


Fig. 1. A nonobservable partial FSM

By direct inspection, one can assure that for this NFSM the Procedure returns the shortest separating sequence of length two and this sequence is $\alpha = i_1 i_2$.

Therefore, separating sequences returned by Procedure 1 can be used for increasing the fault coverage of a transition tour of different EFSM slices. Additional experimental research is needed in order to estimate the fault coverage of transition tours appended with corresponding separating sequences.

IV. CONCLUSION

In this paper, we have focused on some techniques for deriving functional tests based on the EFSM model. The main idea behind the described techniques is to traverse an appropriate set of transitions; this set can be derived as a set of transitions of an FSM obtained by the simulation the EFSM behavior. Experimental results of testing protocol implementations clearly show that the fault coverage of such tests is rather high. In order to enhance the fault coverage, an initial test suite traversing an appropriate set of transitions can be augmented with distinguishing sequences for final states of traversed transitions and we have proposed how such distinguishing sequences can be derived using two FSM slices of the initial EFSM. When deriving distinguishing sequences for FSM slices, we have adapted the known methods for separating states of an observable nondeterministic FSM to FSMs which can be partial and nonobservable.

REFERENCES

- [1] H. Konig, *Protocol Engineering*. Springer, 2012.
- [2] A. Petrenko, S. Boroday, and R. Groz, "Confirming configurations in EFSM testing," *IEEE Trans. Software Eng.*, vol. 30, no. 1, 2004.
- [3] A. Kolomeez, "Algoritmy sinteza proveryayushhikh testov dlya upravlyayushhikh sistem na osnove rasshirenykh avtomatov (in Russian)," Ph.D. dissertation, 2010.
- [4] S. Nica, "On the use of constraints in program mutations and its applicability to testing," Ph.D. dissertation, 2013.
- [5] A. Gill, "State-identification experiments in finite automata," *Information and Control*, pp. 132–154, 1961.
- [6] K. El-Fakih, S. Prokopenko, N. Yevtushenko, and G. von Bochmann, "Fault diagnosis in extended finite state machines," in *Proceedings of the TestCom*, 2003, pp. 197–210.
- [7] A. Faro and A. Petrenko, "Sequence generation from EFSMs for protocol testing," in *Proceedings of the COMNET, Budapest*, 1990, pp. 17–26.
- [8] M. Zhigulin, A. Kolomeez, N. Kushik, and A. Shabaldin, "Testirovanie programmnoj realizatsii protokola irc na osnove modeli rasshirennogo avtomata (in Russian)," *Vestnik Tomskogo politekhnicheskogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika*, pp. 81–84, 2011.

- [9] M. Zhigulin, S. Prokopenko, and M. Forostyanova, "Detecting faults in TFTP implementations using finite state machines with timeouts," in *Proceedings of the SYRCoSE*, 2012, pp. 115–118.
- [10] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, "FSM-based conformance testing methods: A survey annotated with experimental evaluation," *Information & Software Technology*, vol. 52, no. 12, pp. 1286–1297, 2010.
- [11] K. El-Fakih, A. Kolomeez, S. Prokopenko, and N. Yevtushenko, "Extended finite state machine based test derivation driving by user defined faults," in *Proceedings of the ICST*, 2008, pp. 308–317.
- [12] N. Spitsyna, K. El-Fakih, and N. Yevtushenko, "Studying the separability relation between finite state machines," *Softw. Test., Verif. Reliab.*, vol. 17, no. 4, pp. 227–241, 2007.
- [13] M. Gromov, N. Kushik, and N. Yevtushenko, "Razlichayushhie ehksperimenty s neinitsial'nymi nedeterminirovannymi avtomatami (in Russian)," *Vestnik Tomskogo gosudarstvennogo universiteta. Upravlenie, vychislitel'naya tekhnika i informatika*, no. 4, pp. 93–101, 2011.
- [14] M. G. Merayo, M. Nunez, and I. Rodriguez, "Formal testing from timed finite state machines," *Computer Networks*, vol. 52, no. 2, 2008.
- [15] A. Petrenko and N. Yevtushenko, "Testing from partial deterministic FSM specifications," *IEEE Trans. on Computers*, vol. 54, no. 9, pp. 1154–1165, 2005.
- [16] RFC2812, "Internet relay chat: Client protocol," 2000.
- [17] A. Shabaldin, "Constructing a tester for checking student protocol implementations," in *Proceedings of the SYRCoSE*, 2007, pp. 23–29.
- [18] RFC1350, "The TFTP protocol," 1992.
- [19] M. Zhigulin, N. Yevtushenko, S. Maag, and A. R. Cavalli, "FSM-based test derivation strategies for systems with time-outs," 2011, pp. 141–149.
- [20] RFC1939, "Post office protocol - version 3," 1996.
- [21] U. Mihailov, "Razrabotka metoda sinteza proveryayushhikh testov dlya rasshirenykh avtomatov na osnove srezov (in Russian)," Master's thesis, 2008.
- [22] A. Nikitin and N. Kushik, "On EFSM-based test derivation strategies," 2010, pp. 116–119.
- [23] J.-H. R. Jiang, C.-C. Lee, A. Mishchenko, and C.-Y. R. Huang, "To SAT or not to SAT: Scalable exploration of functional dependency," *IEEE Trans. Computers*, vol. 54, no. 9, pp. 457–467, 2010.
- [24] K. L. McMillan, "Interpolation and SAT-based model checking," in *Proceedings of the CAV*, 2003, pp. 1–13.
- [25] A. Solodovnikov, *Systems of Linear Inequalities*. Popular Lectures in Mathematics, 1980.
- [26] N. Kushik and N. Yevtushenko, "On the length of homing sequences for nondeterministic finite state machines," in *Proceedings of the CIAA*, 2013, pp. 220–231.
- [27] D. Lee and M. Yannakakis, "Testing finite-state machines: State identification and verification," *IEEE Transactions on Computers*, vol. 43, no. 3, 1994.
- [28] J. Kim and M. Newborn, "The simplification of sequential machines with input restrictions," *IEEE Transactions on Computers*, vol. 21, no. 12, 1972.
- [29] A. Petrenko, N. Yevtushenko, and R. Dssouli, "Testing strategies for communicating FSMs," in *Proceedings of the IFIP Seventh International Workshop on Protocol Test Systems*, 1994, pp. 193–208.

A generic knowledgebase for test generation

Artem Kotsynyak, Andrei Tatarnikov

Software Engineering Department

Institute for System Programming of the Russian Academy of Sciences (ISPRAS)

Moscow, Russian Federation

Email: {kotsynyak, andrewt}@ispras.ru

Abstract— Nowadays a lot of various test generation tools are developed and applied to create tests for both software applications and hardware designs. Taking into account the size and complexity of modern projects, there is an urgent need for "smart" tools that would help maximize test coverage and keep the required effort and time to a minimum. Despite the fact that each project is unique in some sense, there is a set of common generation techniques that are applied in a wide range of projects (random tests, combinatorial tests, tests for corner cases, etc). In addition, projects belonging to specific domains tend to share similar test cases or use similar heuristics to generate them. A natural way to improve the quality of testing is to make the most of the experience gained working on different projects or performing testing at different stages of the same project. To achieve this goal, a knowledgebase holding information relevant to test generation would be of a great help. This would facilitate reuse of test cases and generation algorithms and would allow sharing knowledge of "interesting" situations that can occur in a system under test. The paper proposes a concept of a knowledgebase for test generation that can be used in a wide range of test generation tools. At ISPRAS, it is applied in test program generation tools that create test programs for microprocessors. The knowledgebase is designed to store information on widely used test generation techniques and test situations that can occur in a microprocessor design under verification.

Keywords— *test generation, testing knowledge, test reuse, test situations, constraints, knowledgebase.*

I. INTRODUCTION

To start with, it should be said that our team works in the area of hardware verification [1, 2]. Therefore, the main focus of the research is on generating tests for hardware devices. However, the concepts described in this paper are not limited to hardware verification and remain relevant to a wide range of domains.

Testing accounts for up to 70% of overall project resources. To reduce the expenses, an effort is made to automate the testing process. Over the recent couple of decades, approaches to automated testing have evolved significantly. Still, increasing complexity of modern projects demands for more efficient methods. To get the big picture of the state of the art, let us first consider existing approaches from the most trivial to the most advanced.

The most straightforward way to automate test generation for one's project is to write a simple test generator in one of the popular programming languages. Such generators are usually

targeted at producing random or combinatorial tests. However, they can also include heuristics that help generate tests for some "interesting" situations (e.g. boundary conditions). This approach has the following obvious disadvantages: such tools are inflexible and the knowledge they include is unsuitable for reuse as it is usually hardcoded. Moreover, random and combinatorial tests are not systematic and cannot guarantee a sufficient level of test coverage.

To cover nontrivial cases that are unreachable by using random and combinatorial generation, a test generation tool should be strengthened to be able to create directed tests [3]. Directed tests are usually generated on the basis of test templates that provide abstract high-level descriptions of testing problems. Such an approach is called template-based generation. Test templates use constraints to formulate conditions of occurrence for situations to be covered. Briefly speaking, constraints are a set of formulae describing relations between data (i.e. properties to be held for some events to fire). One of the advantages of template-based generation is that it separates test generation logic from the description of specific test cases, which simplifies test maintenance. More importantly, this allows constraints to be reused in other tests. However, the reuse is limited as constraints are described in terms of the verified system (hardware design or software application) and are not systematized. The issue is that manual creation of complex constraints is quite laborious and it might require a significant effort to adapt them for a different system. In fact, this could be improved as constraints for similar verification tasks tend to share common parts.

The next step in the evolution of approaches to test generation is model-based generation [1, 2, 3]. It implies separation of knowledge of the verified system's configuration from knowledge of test generation techniques. The former is referred to as a model and the latter is often called testing knowledge [5, 6]. The model can be created either manually or built automatically on the basis of formal specifications. The advantage of this approach is that it allows describing test cases in terms of the model, which results in more abstract descriptions. In addition, the model often includes coverage information that can be extracted from formal specifications or from other sources. In a nutshell, to generate high-quality tests, two types of knowledge are required: (1) knowledge of the verified system's configuration to be able to generate valid tests and (2) knowledge about situations that can occur in the system to be able to generate tests that would hit all "corners" of that system. Coverage information can be represented by a set of constraints describing conditions for various test situations. In this case, to generate test programs for the target system, one needs to provide a test template specified in terms

of information exposed by the model and constraints describing corresponding situations. As it can be noticed, to provide a good quality of test coverage, it may require creating a significant amount of test templates describing test cases for all possible situations. When this job is done manually, it can be time-consuming and there is a chance to miss some "interesting" cases especially when the constraints are not systematized and the coverage model changes as new knowledge about the system is acquired.

To further automate the process of test program generation, constraints need to be stored in a systematized way. In other words, knowledge of "interesting" situations and knowledge of how to obtain data causing these situations to fire should be accumulated in a knowledgebase for further use in the test generation process. Also, it would be highly desirable to have this information stored in a human-readable form to simplify its reuse and the maintenance of the knowledgebase. This leads to an idea of a knowledgebase that would store testing knowledge including commonly used constraints, algorithms for solving them, algorithms of random and combinatorial test data and test sequence generation, methods of exploring properties of the verified system's model, etc. Having this knowledge stored in a systematized way will allow making more intelligent decisions during test generation. One of the main goals is to reduce the number of test templates. The use of the knowledgebase would allow creating some of them in an automated way, therefore reducing the effort and increasing the coverage quality. Also, having a centralized store of testing knowledge gives a great advantage in terms of reuse and sharing experience between test engineers.

As the project the verification team is working on moves from the requirement elicitation to the release, more and more testing knowledge is accumulated. It may come from different sources such as requirements, specification, expertise, failed tests, automated analysis, etc. Some of this knowledge can be presented in an abstract way so that common test cases like overflows and other could be reused in projects with similar components. A centralized store helps ensure that each test engineer has this knowledge in hand and no "interesting" situation is ignored.

The present paper describes concepts of a knowledgebase for test generation. The knowledgebase is being developed at ISPRAS to be used in projects dedicated to hardware verification [1, 2].

The rest of the paper is organized as follows. Section II gives an overview of existing works related to testing knowledge. Section III provides a list of core requirements for the knowledgebase. Section IV describes the architecture of the knowledgebase and explains how it can be integrated with test generation tools. Finally, Section V concludes the paper.

II. RELATED WORK AND MOTIVATION

Methods of efficient test generation have always been a major subject of research. One of the most important applications is functional verification of microprocessors where test program generation and simulation is the most common approach applied at the system level. Due to enormous complexity of modern microprocessors and severe time-to-

market pressures, it is quite a challenging task. For this reason a lot of effort has been invested to maximize automation of this activity. This resulted in the emergence of a great number of test generation techniques. Also, a significant amount of knowledge about bug-prone areas in hardware designs has been accumulated. An important direction is to systematize the accumulated knowledge to further automate the test generation process and reduce its cost by facilitating knowledge reuse.

IBM Research [3, 5, 6] has been one of the main contributors in the field of test program generation for microprocessors during the last decades. The first test generation tools were developed in the middle of 1980s. Test program generators by IBM Research have evolved over time from random to directed model-based generation schemes. Genesys-Pro, one of the most recent tools, uses test templates that describe test generation problems as constraint satisfaction problems and uses a generic constraint solver customized for pseudorandom generation to increase the coverage quality. Constraints are based on the architectural description captured by the model and on the testing knowledge representing a set of methods that help increase the quality of generated test cases. There are two types of constraints: (1) mandatory ("hard") and (2) non-mandatory ("soft"). Constraints that originate from architectural description are typically marked as mandatory. "Soft" constraints help shift the bias of the generated stimulus to make test cases more "interesting" and can be ignored if the solver fails to find a solution. Testing knowledge, as it is described in papers by IBM Research, represents a collection of architecture-independent constraints and constraints specific to a given design. Also, it includes a set of heuristics that use accumulated knowledge of the semantics of the verified design to shift bias towards specific constraints to maximize coverage. IBM Research does not reveal details on how exactly the storage of testing knowledge is organized and integrated with their tools. However, their testing knowledge is obviously oriented only towards test program generation for microprocessors and is likely to be tightly coupled with their test generation tools. Two important aspects that were not covered in their papers are: (1) systematization of constraints and (2) means of combining constraints to describe complex problems (this particularly applies to constraints of different types). It is possible to specify probability distributions between "soft" constraints in a test template. However, there are reasons to think that no facilities are provided to do this at the level of testing knowledge.

Another company that has made a significant contribution in development of test generation tools is Obsidian Software (now acquired by ARM) [4]. The company specializes in development of verification and validation software used in the design of microprocessors. Their test program generation tool RAVEN (Random Architecture Verification Engine) is able to generate random and directed tests based on test templates. To achieve a better coverage, it makes use of coverage grids and accumulates verification knowledge in a database. Test templates are focused on the coverage grid and use constraints that allow RAVEN to intelligently choose random values to reach specific coverage goals. Unfortunately, documentation available on the tool does not provide detailed information on how the mechanism of knowledge accumulation is organized.

The motivation of the present research is to work out the concepts and to design the architecture of a knowledgebase for test generation that could be used in a wide range of test generation tools. It should help systematize various types of testing knowledge and facilitate its accumulation and reuse. The paper aims to contribute to the research in the field as the lack of information on competitors' solutions makes it difficult to apply their ideas. The paper summarizes the ideas from different sources [4, 5, 6], proposes some important improvements and expresses our vision for organization of a knowledgebase for test generation.

III. REQUIREMENTS FOR THE KNOWLEDGEBASE

The knowledgebase should maximize the quality of test coverage and minimize the effort required to create tests. For this purpose it accumulates knowledge about different test situations (conditions that make them fire, probabilities of their occurrence, methods of producing corresponding stimuli, etc.) This creates a possibility to easily create complex test cases by combining the accumulated knowledge. If this job is automated, it will help reduce the number of test cases described manually, therefore increasing the productivity of the verification team. Here is the list of the main requirements a generic knowledgebase for test generation should satisfy to achieve its goals:

- 1) The knowledgebase should be able to store and accumulate testing knowledge of a wide range of types coming from various sources and having different formats. This includes sets of test values, commonly used generation algorithms, constraints, methods of combining them, heuristics for shifting biases, etc.
- 2) The stored knowledge should be systematized and organized into a hierarchy. This will simplify its maintenance and reuse and will allow extracting common components.
- 3) It should be possible to easily integrate the knowledgebase into test generation environments of different kinds. The client environment should be provided with full access to the accumulated knowledge. To facilitate it, the knowledgebase should be implemented as an open-source project.
- 4) The knowledgebase should facilitate the transfer of project-independent knowledge between projects in a similar domain. This applies to test situations, constraints, data generators, etc.

IV. ARCHITECTURE

The most important components of the knowledgebase architecture are the *storage engine*, *selector* and *resolution module* as shown in Figure 1. Further in this section, they will be discussed in more detail.

The job of the *storage engine* is to provide a persistent storage for any kind of knowledge allowed. The storage engine can be powered by any database technology. To add new knowledge or alter existing, users should interact with the engine via the *control interface* built on top of it. The interface provides access to logical representation of the stored

knowledge hiding any details about the underlying database and data organization along with normalization logic specific to the knowledgebase implementation.

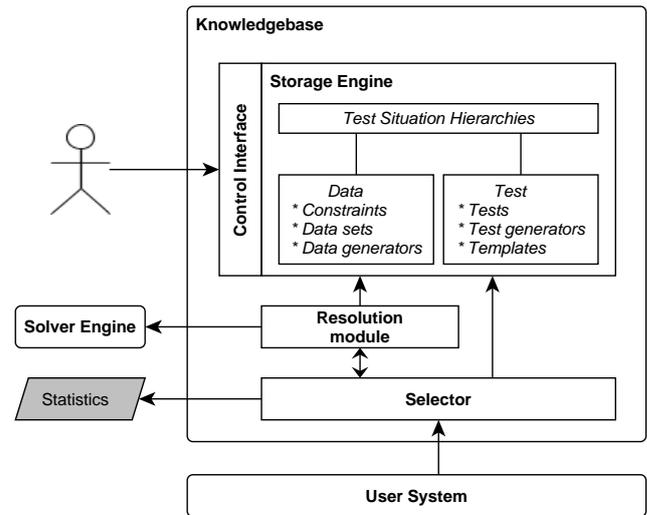


Figure 1. Architecture scheme

The main responsibility of the knowledgebase is to store hierarchies of test situations along with associated knowledge. Hierarchies of test situations being part of testing knowledge themselves are used as a tool for organizing the acquired knowledge. Therefore, it is up to the storage engine to store, manage and provide access to these structures to the rest of the knowledgebase modules. Nevertheless, the storage engine treats knowledge as data and does not implement any additional logic beyond what is encapsulated in the control interface.

Basically, a test situation in a hierarchy is a symbolic representation of an event (or a group of events) that can occur in the system under test, but the hierarchy itself does not provide any information about what kind of event it is. Hierarchies are represented by directed acyclic graphs (DAG) where nodes specify test situations while arcs denote refinement relation, i.e. if there is a path from node u to node v then node v represents a situation that is a special case of the situation of node u .

There are two types of situation hierarchies: *abstract* and *concrete*. Abstract hierarchies are used to describe commonalities between projects belonging to the same domain, while concrete hierarchies specify relations between particular test cases. Representations of abstract and concrete hierarchies have several important differences. First, abstract hierarchies are represented by unweighted DAGs, while weighted DAGs are used for concrete hierarchies where arc weights denote the desired probabilities of corresponding events. Second, nodes in a concrete hierarchy can be associated with additional knowledge about situations represented by these nodes (e.g. constraints describing the conditions for corresponding events to fire).

Figure 2 shows an example of a simple situation hierarchy that specifies situations from the microprocessor verification domain. The "add" node denotes situations possible in the

execution flow of an addition instruction and the “overflow” node denotes any kind of an overflow situation. Also, there is a refinement for an integer overflow called “int_overflow” and two explicit terminal situations called “corner” and “zero_sum” (the former describes a corner case for the integer overflow situation and the latter specifies the zero-sum situation). It is shown that the “int_overflow”, “corner” and “zero_sum” situations are associated with constraints describing data resulting in corresponding events. Implicit situations for the normal flow and random values are omitted along with the probabilities of their occurrence.

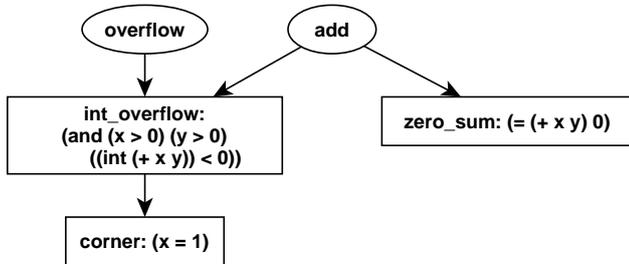


Figure 2. Example of a situation hierarchy

Arc weights in a concrete hierarchy describe relative rates at which specific test situations are to be obtained during test generation. Therefore, this can be used to control the generation process since it is allowed to bias probabilities in order to get behavior varying from fully random to fully deterministic. Moreover, zero probability effectively removes test situations from being exploited in test generation and can be used to disable unimplemented or irrelevant features of the current system under test.

Associated knowledge in concrete hierarchies may vary from complete tests to abstract templates. It can be stored in a database or in plain files. One of our goals is to reuse existing test generators so it is allowed to use them as associated knowledge via appropriate adaptors. This knowledge is primarily used in the test generation process and the querying system should be able to handle it by itself.

The control interface of the storage engine provides mostly database editor functionality. Therefore, to handle queries to the knowledgebase with respect to knowledge semantics, a specific module called *selector* has been introduced. The module “knows” about knowledge organization and uses probabilities stored within concrete hierarchies to select specific knowledge. Since every query to the knowledgebase is passed through the selector, it can trace test situations queried and produce a statistical report that can be used to adjust situation probabilities or to perform coverage analysis.

In the simplest cases, the selector just fetches the stored data and passes it to the querying system. This works for flat data and tests, but not for generators and constraints. To handle these correctly, an additional component called the *resolution module* has been included. Its initial purpose is to run generators stored as knowledge or to pass constraints to some external solver to produce data. The resolution module is designed to be an extension mechanism that has access to all internals of the knowledgebase and is allowed to run external

applications. It is used whenever the selector decides that knowledge requires additional treatment before being sent to the querying system. Therefore, it can be adjusted in a domain-specific way to handle much more sophisticated scenarios, e.g. generate tests on the fly if a test template has been queried.

It should be noted that despite the fact that the knowledgebase considers test generators and tests as knowledge it is not a test generation system. Generating tests using the knowledgebase is straightforward and can be done at different levels depending on the contained knowledge and its organization, e.g. in microprocessor verification we can potentially generate test data for a single instruction, for a complex test template or generate a final test program using stored tests and generators. The test generation system queries the knowledgebase for test situations that correspond to terminal or non-terminal nodes in the hierarchy. In the latter case, the selector will use some refinement of the situation given with respect to probabilities stored within the hierarchy. Either the selector is able to fetch knowledge by itself, or it delegates the task to the resolution module, or both of them fail because the storage engine does not contain knowledge required or resolution marks the query unsatisfiable. In a successful scenario, the output is a test or test data and it is up to the querying system to distinguish between them.

V. CONCLUSION

We have proposed the concept and the architecture of a generic knowledgebase for test generation. The knowledgebase can be used in a wide range of test generation tools to accumulate knowledge related to the system under test. At ISPRAS, it will be integrated with tools responsible for test program generation for microprocessors. It facilitates knowledge reuse and allows making “smart” decisions during the process of test generation based on the accumulated knowledge. This helps improve test coverage and simplify test development.

REFERENCES

- [1] A. Kamkin and A. Tatarnikov, MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors, proceedings of the 6th Spring/Summer Young Researchers’ Colloquium on Software Engineering (SYRCoSE 2012), 2012, pp. 64-69.
- [2] A. Kamkin, T. Sergeeva, A. Tatarnikov and A. Utekhin, MicroTESK: An Extendable Framework for Test Program Generation, proceedings of the 7th Spring/Summer Young Researchers’ Colloquium on Software Engineering (SYRCoSE 2013), 2013, pp. 51-57.
- [3] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov and A. Ziv, Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification, IEEE Design & Test of Computers, 2004, pp. 84-93.
- [4] <http://www.obsidiansoft.com/pdf/Datasheet.pdf>
- [5] L. Fournier, Y. Arbetman, and M. Levinger, “Functional Verification Methodology for Microprocessors Using the Genesys Test Program Generator: Application to the x86 Microprocessors Family,” Proc. Design Automation and Test in Europe (DATE 99), IEEE CS Press, 1999, pp. 434-441.
- [6] R. Emek, I. Jaeger, Y. Katz and Y. Naveh, “Quality Improvement Methods for System-level Stimuli Generation”, Proc. Computer Design: VLSI in Computers and Processors (ICCD 2004), IEEE CS Press, 2004, pp. 204-206.

Interactive test case design via attribute exploration

Fedor Strok
Yandex, NRU-HSE
fdr.strok@gmail.com

Georgy Kondratiev
Yandex
orivej@gmx.fr

Abstract—Techniques of test case design usually either rely on expert manual work or employ algorithmic approaches like pairwise testing. However, pairwise testing has several known limitations: implicit relation between test cases and the domain, complexity of oracle implementation. Interactive test case design with attribute exploration is a technique that unites best practices in a semi-supervised procedure to explore the domain and generate test cases. It is based on well-studied algorithms of Formal Concept Analysis.

Keywords—formal concept analysis, pairwise testing, attribute exploration.

I. INTRODUCTION

Software testing usually aims at the quality assurance of software. One of its major goals is to properly describe and consider all possible test cases. One of the most widely used approaches, Domain Testing, singles out the main parameters that influence the output, and checks their possible value combinations.

Enumerating all value combinations leads to exponential complexity: just six boolean parameters produce 64 combinations, already too much for an expert to evaluate unmistakably. During manual test case design, experts choose some reasonable subset of combinations, bearing the risk to leave important combinations uncovered, which increases with the number of parameters.

Pairwise testing [2] and its generalization, n-wise testing, became popular in automating this process. We define parameters and domains and pass them as a model to a black-box algorithm [1] that produces a set of test cases, guaranteed to cover all combinations of each pair (or n-tuple, as specified in the model) of input parameters. Generating the optimal test set is computationally hard, so it is approximated with some randomized heuristic algorithms. This makes generated cases different from run to run, unless random seed is fixed. The main advantage of this approach is its little sensitivity to the number of parameters.

Typically, input parameters are not completely independent. A formal way to express dependencies is implications: statements in the form ‘if ..., then ...’. Consideration of parameter interdependence decreases quantity of resulting cases by excluding some of possible combinations.

Our approach is focused on implications. We use the algorithms of Formal Concept Analysis to provide software engineers with a tool that helps explore the domain in a semi-automatic way. It guarantees sound and complete description if the expert gives valid answers to the system.

The rest of the paper is structured as follows. Section 2 introduces basic notions of Formal Concept Analysis. Section 3

focuses on the procedure of attribute exploration. Section 4 provides examples of attribute exploration in the fields of numbers and logic formulae. Section 5 is the conclusion.

II. FORMAL CONCEPT ANALYSIS

Formal Concept Analysis [4] is a technique introduced by Rudolf Wille in 1984 to derive a formal ontology from a collection of objects and attributes. It relies on lattice and order theories [3]. Numerous applications are found in the field of machine learning, data mining, text mining and biology.

A *formal context* is a triple (G, M, I) , where G is a set of objects, M is a set of attributes, and $I \subseteq G \times M$ is a binary relation between G and M . In other words, for objects in G there exists a description in terms of attributes in M , and relation I reflects that an object has an attribute: $(g, m) \in I$ means that object g possesses m .

Here is an example of a formal context:

$G \setminus M$	a	b	c	d
	×			×
	×		×	
		×	×	
		×	×	×

Objects:

- 1 – equilateral triangle
- 2 – right triangle
- 3 – rectangle
- 4 – square

Attributes:

- a – 3 vertices
- b – 4 vertices
- c – has a right angle
- d – all sides are equal

Consider two mappings for a given context:

$$\varphi: 2^G \rightarrow 2^M \quad \varphi(A) \stackrel{\text{def}}{=} \{m \in M \mid gIm \text{ for all } g \in A\}$$

$$\psi: 2^M \rightarrow 2^G \quad \psi(B) \stackrel{\text{def}}{=} \{g \in G \mid gIm \text{ for all } m \in B\}$$

For all $A_1, A_2 \subseteq G, B_1, B_2 \subseteq M$

- 1) $A_1 \subseteq A_2 \Rightarrow \varphi(A_2) \subseteq \varphi(A_1)$
- 2) $B_1 \subseteq B_2 \Rightarrow \psi(B_2) \subseteq \psi(B_1)$
- 3) $A_1 \subseteq \psi\varphi(A_1) \quad B_1 \subseteq \varphi\psi(B_1)$

Traditionally, notation $(\cdot)'$ is used instead of φ and ψ . $(\cdot)''$ stands for $\varphi \circ \psi$ or $\psi \circ \varphi$ (depending on its argument). Thus, for arbitrary $A \subseteq G, B \subseteq M$:

$$A' \stackrel{\text{def}}{=} \{m \in M \mid gIm \text{ for all } g \in A\},$$

$$B' \stackrel{\text{def}}{=} \{g \in G \mid gIm \text{ for all } m \in B\}.$$

(Formal) concept is a pair (A, B) : $A \subseteq G$, $B \subseteq M$, $A' = B$, $B' = A$.

In the example with geometric figures, a pair $(\{3, 4\}, \{b, c\})$ is a formal concept. For a formal context (G, M, I) , $A, A_1, A_2 \subseteq G$ — sets of objects, $B \subseteq M$ — a set of attributes, the following statements hold for operation $(\cdot)'$:

- 1) $A_1 \subseteq A_2 \Rightarrow A'_2 \subseteq A'_1$,
- 2) $A_1 \subseteq A_2 \Rightarrow A'_1 \subseteq A'_2$
- 3) $A \subseteq A''$
- 4) $A''' = A'$ and $A'''' = A''$
- 5) $(A_1 \cup A_2)' = A'_1 \cap A'_2$
- 6) $A \subseteq B' \Leftrightarrow B \subseteq A' \Leftrightarrow A \times B \subseteq I$

Closure operator on set G is a mapping $\gamma: \mathcal{P}(G) \rightarrow \mathcal{P}(G)$, which maps every $X \subseteq G$ to the closure $\gamma X \subseteq G$ under the following conditions:

- 1) $\gamma\gamma X = \gamma X$ (idempotence)
- 2) $X \subseteq \gamma X$ (extensivity)
- 3) $X \subseteq Y \Rightarrow \gamma X \subseteq \gamma Y$ (monotonicity)

Implication $A \rightarrow B$, where $A, B \subseteq M$, takes place if $A' \subseteq B'$, in other words, if each object having A also has all attributes from B .

III. ATTRIBUTE EXPLORATION

Attribute exploration is well known within Formal Concept Analysis. Its general idea is to explore the object domain semi-automatically. This means that an expert is necessary, but his duty is to answer specific questions about possible dependencies in the area. Questions are presented as implications either to confirm or to decline. If confirmed, an implication is added to the base of knowledge. If declined, the expert is asked to provide a counterexample violating proposed dependency.

In other words, exploration algorithm wants to explore all possible combinations of a given attribute set. Since it is typical that objects are too difficult to enumerate, the algorithm starts with a small set of examples. Then it computes canonical base of implications for the provided formal context. Then it asks the domain expert if the computed implications are valid in general. If so, existing context represents all possible combinations in the domain. Otherwise, there exists a counterexample in the domain, which is added to the context, and canonical base is recalculated.

General strategy is quite intuitive: after enumerating all relevant attributes, we start exploring the domain with some knowledge of typical examples and dependencies. To extend knowledge database we either add a rule, or provide another example that violates currently standing dependencies.

Algorithm 1 NEXT CLOSURE(A, M, \mathcal{L})

Input: Closure operator $X \mapsto \mathcal{L}(X)$ on an arbitrarily ordered set M and a subset $A \subseteq M$.

Output: *lectically* next closed set after A .

```

for all  $m \in M$ , in order, do
  if  $m \in A$  then
     $A := A \setminus \{m\}$ 
  else
     $B := \mathcal{L}(A \cup \{m\})$ 
    if no element in  $B \setminus A$  comes after  $m$  in  $M$  then
      return  $B$ 
return  $\perp$ 

```

Algorithm 2 ATTRIBUTE EXPLORATION

Input: A subcontext $(E, M, J = I \cap E \times M)$ of (G, M, I) , possibly empty.

Input: Interactive: confirm that $A = B''$ in a formal context (G, M, I) , M finite, or give an object showing that $A \neq B''$.

Output: The canonical base \mathcal{L} of (G, M, I) and a possibly enlarged subcontext $(E, M, J = I \cap E \times M)$ with the same canonical base.

```

 $\mathcal{L} := \emptyset$ 
 $A := \emptyset$ 
while  $A \neq M$  do
  while  $A \neq A^{JJ}$  and  $A^{JJ} \neq A^{II}$  do
    extend  $E$  by some object  $g \in A^I \setminus A^{JJI}$ 
  if  $A^{JJ} = A^{II}$  then
     $\mathcal{L} := \mathcal{L} \cup \{A \rightarrow A^{JJ}\}$ 
     $A := \text{NextClosure}(A, M, \mathcal{L})$ 
return  $\mathcal{L}, (E, M, J)$ 

```

IV. INSTRUCTIVE EXAMPLES

A. Numbers

Let us consider the domain of natural numbers [5]. As the set of possible attributes we choose the following: even ($2*n$), odd ($2*n+1$), divisible-by-three ($3*n$), prime (has no positive divisors other than 1 and itself), factorial (is a factorial of a positive number). We start from an empty set of objects. The canonical base for such context is $\emptyset \rightarrow M$. We get a question:

Is the following implication valid?
 \Rightarrow even, factorial, divisible-by-three, odd, prime

Obviously, not all numbers possess all attributes. Let's start with 2, which is even, factorial, prime. We add 2 to our context and recalculate the base.

$G \setminus M$	even	factorial	divisible-by-three	odd	prime
2	×	×			×

Is the following implication valid?
 \Rightarrow even, factorial, prime

Now we can think of number 5, which is prime, odd.

$G \setminus M$	even	factorial	divisible-by-three	odd	prime
2	×	×			×
5				×	×

Is the following implication valid?
 \Rightarrow prime

Now we are about to either state that all numbers are prime, or provide a non-prime number, e.g. **6**.

Is the following implication valid?
factorial \Rightarrow even

Now we have entered 2 and 6, which are simultaneously even and factorial. To give a counterexample we should find an odd factorial, which is **1**.

Is the following implication valid?
odd \Rightarrow prime

That does not hold for number **9**.

Is the following implication valid?
factorial, odd \Rightarrow prime

There is only one odd factorial, 1, and it is prime, so we confirm this implication.

Is the following implication valid?
factorial, divisible-by-three \Rightarrow even

The least factorial divisible by three is 6, and it is even. All larger factorials divisible by three, being multiples of the least one, are also even.

Is the following implication valid?
prime, divisible-by-three \Rightarrow even, *factorial, odd*

There is **3**, which is just odd.

Is the following implication valid?
prime, divisible-by-three \Rightarrow odd

The only prime divisible by three is three itself, so the implication is true.

Is the following implication valid?
even \Rightarrow factorial

Not all even numbers are factorials, e.g. **8**.

Is the following implication valid?
even, odd \Rightarrow factorial, *prime, divisible-by-three*

No numbers are both even and odd, so the premise is false and the implication is true.

Is the following implication valid?
even, divisible-by-three \Rightarrow factorial

There is **12**, which is even and divisible by three, but it is not a factorial.

Is the following implication valid?
even, prime \Rightarrow factorial

The only even prime number 2 is a factorial.

And the exploration process is over. The final context:

G \ M	even	factorial	divisible-by-three	odd	prime
2	×	×			×
5				×	×
6	×	×	×		
1		×		×	×
9			×	×	
3			×	×	×
8	×				
12	×		×		

The set of implications:

- *factorial, odd* \Rightarrow prime
- *factorial, divisible-by-three* \Rightarrow even
- *prime, divisible-by-three* \Rightarrow odd
- *even, odd* \Rightarrow factorial, *prime, divisible-by-three*
- *even, prime* \Rightarrow factorial

B. Logic functions

It is inherent in our approach that dependencies involve only presence of attributes, not their absence. To cover a boolean variable, we have to introduce two attributes, one for its truth and one for falsehood. This leads to unnecessary questions during exploration, such as $A \Rightarrow \neg A$? To eliminate them, we automatically assign a value to an attribute whenever its opposite is set. However, we could still face implications of the form $A \wedge \neg A \wedge \dots \Rightarrow \dots$?, which we automatically accept, as it covers an impossibility.

Let us consider the model situation when we are testing a logical function $(a \wedge b) \rightarrow c$. This example may seem unnatural, but it shows an application of the main principles.

1) $\neg a \wedge a \wedge \neg b \wedge b \wedge \neg c \wedge c \wedge \neg result \wedge result$? No

	a	b	c	result
1	False	False	True	True

2) $\neg a \wedge \neg b \wedge c \wedge result$? No

2	False	False	False	True
---	-------	-------	-------	------

3) $\neg a \wedge \neg b \wedge result$? No

3	True	True	False	False
---	------	------	-------	-------

4) $a \Rightarrow b \wedge \neg c \wedge \neg result$? No

4	True	False	False	True
---	------	-------	-------	------

5) $a \Rightarrow \neg c$? No

5	True	False	True	True
---	------	-------	------	------

6) $b \Rightarrow a \wedge \neg c \wedge \neg result$? No

6	False	True	True	True
---	-------	------	------	------

7) $a \wedge b \Rightarrow \neg c \wedge \neg result$? No

7	True	True	True	True
---	------	------	------	------

- 8) $c \Rightarrow result$? Yes
 9) $b \wedge result \Rightarrow c$? No

	a	b	c	result
1	False	False	True	True
2	False	False	False	True
3	True	True	False	False
4	True	False	False	True
5	True	False	True	True
6	False	True	True	True
7	True	True	True	True
8	False	True	False	True

- 10) $a \wedge b \wedge result \Rightarrow c$? Yes
 11) $\neg a \Rightarrow result$? Yes
 12) $\neg b \Rightarrow result$? Yes
 13) $a \wedge b \wedge \neg c \Rightarrow \neg result$? Yes
 14) $a \wedge \neg c \wedge result \Rightarrow \neg b$? Yes
 15) $b \wedge \neg c \wedge result \Rightarrow \neg a$? Yes
 16) $\neg result \Rightarrow a \wedge b \wedge \neg c$? Yes

Thus, our procedure examines all 16 possible value combinations. If we skip rules with 'result' in the premise, the final rules are:

- $c \Rightarrow result$
- $\neg a \Rightarrow result$
- $\neg b \Rightarrow result$
- $a \wedge b \wedge \neg c \Rightarrow \neg result$

V. CONCLUSION

Mathematical toolbox of Formal Concept Analysis provides interactive procedure for arbitrary domain description. We apply this methodology to the problem of test case design. This preserves the major advantage of manual test case design: each test case breaks an improper implication, so each test case has a reason to be maintained. But unlike manual work, it guarantees full coverage, i.e. completeness of implications and counter-examples, if all questions were answered correctly.

One of the main advantages of interactive test case design is dual nature of its output. Besides test cases it generates implications that could be used for reference and documentation purposes.

Another advantage of the proposed technique is its extensibility. If we add a new attribute, we just initialize a new formal context with the previous examples (assuming that the new attribute is absent from all objects, and refining examples where this assumption does not hold), and proceed with attribute exploration. Former implications then may also be reused.

Proposed algorithm is usable not only as a standalone solution for test case design, but also as a tool to discover existing dependencies in the domain. Obtained implications could be valuable in pairwise testing to adjust the model.

However, the current approach is limited in relation to attribute descriptions. For now, it is highly dependent on the boolean nature of attributes. One of the main directions of future work is to support attributes of arbitrary types.

REFERENCES

- [1] Bach, J. and Shroeder, P. 2004. Pairwise Testing A Best Practice That Isn't. In Proceedings of the 22nd Pacific Northwest Software Quality Conference, 2004.
- [2] Czerwonka, J. 2006. Pairwise Testing in Real World: Practical Extensions to Test Case Generators. In Proceedings of the 24th Pacific Northwest Software Quality Conference
- [3] Davey, B. and Priestly, H. *Introduction to Lattices and Order (2nd edition)*. Cambridge Mathematical Textbooks. Cambridge University Press, 2002.
- [4] Ganter, B. and Wille, R., *Formal Concept Analysis: Mathematical Foundations*, Springer, 1999.
- [5] <https://github.com/ae-hse/fca/>

Keyword-Driven Testing with Message Sequence Charts

Boris Tyutin, Alexey Veselov, Vsevolod Kotlyarov

Saint Petersburg State Polytechnical University, Saint Petersburg, Russia
b.tyutin@ics2.ecd.spbstu.ru, veselov.alexey@gmail.com, vpk@ics2.ecd.spbstu.ru

Abstract — This paper overviews an approach to keyword-driven testing based on test cases created in Message Sequence Charts format. Main features and advantages of this idea are discussed. Last two chapters provide brief overview of current implementation of testing automation framework based on the presented approach.

I. INTRODUCTION

Nowadays software development includes a wide range of techniques and strategies. During the past years they evolved from heavy and strict methodologies like waterfall life cycle to agile techniques and iterative approaches. All of them are now more or less standardized and are used in different areas of software engineering. The choice of the development model is driven by the characteristics of the particular projects.

In all kind of development processes we can find phases that have some particular goal and go one after another. In iterative approaches it is possible to get back to some of the previous step if something goes wrong. And testing is a reasonable approach for checking whether the whole work is done well.

It is obvious that different types of workflow activities require different types of testing. Being in a stage of requirement clarifying we can operate only a model of future software. At that moment it is impossible to do performance testing. Moreover, it is not required as the main goal of that stage is to reveal contradictions and gaps in the specification. But it is extremely expensive to maintain different testing processes for one product. It is required not only to integrate different technologies but also to maintain multiple test suites and keep them coherent with the requirements. To reduce costs testing should be scalable and allow applying the same approach on different life cycle stages.

Another problem of testing is that traditionally it is directed towards the engineers. Being unavailable for business people, information about performed checks and probations of software product becomes less useful for planning or marketing. And, vice versa, making test results more obvious for most of the stakeholders of developing process increases benefits of testing. So it can be used not only for finding bugs but also to collect and maintain that database of knowledge about the product. Test-driven development [1] is a good example of an attempt to achieve this goal.

Among the variety of existing testing approaches keyword-driven testing (KDT) is one of the most advanced [2]. It aims at simplifying the test suite development and maintenance and separates the logic of test procedures from the implementation. This paper describes an approach for automated testing based on Message Sequence Charts (MSC) [3] which implements KDT approach. Main concepts of the latter one are briefly overviewed and compared with the capabilities of MSC format. Basing on both concepts a testing approach is developed, and two main ways of its implementation are described. The article is concluded with the overview of current results and future plans.

II. KEYWORD-DRIVEN APPROACH

KDT is a third-generation approach for automated testing framework design. This means that it allows creating and executing structured scenarios with data being separated from control flow. Keyword-driven tests consist of a list of keywords and their parameters. Each keyword represents a predefined set of actions performed against the system under test (SUT). The scenario itself has tabular format, and can be edited as plain text or with special tools. Fig. 1 represents the high-level concepts of KDT approach.

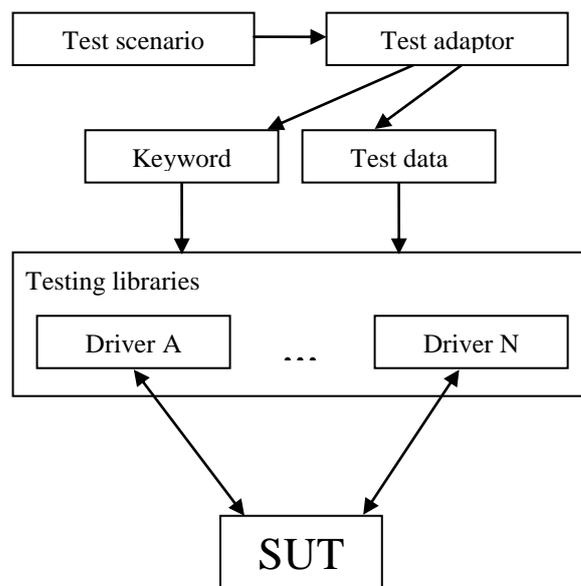


Figure 1. Keyword-driven testing.

Each keyword has clear and unambiguous definition. It can be implemented in code or written in document. Thereby KDT can be applied both for automated and manual testing. It is possible to develop keyword definitions separately from the tests, and, thus, testing can be divided into two independent flows – test design and test action specification. The latter one can be done by someone without programming skills.

With KDT test suite becomes more stable. The cost of its maintenance is reduced because it is not necessary to fix test scenarios according to the changes in SUT, only keyword definitions. Test execution is more scalable [4]. Test cases itself become easy to modify as they operate high-level abstractions and can reuse existing keywords. Due to the absence of low-level details test suite is readable by all stakeholders.

III. MESSAGE SEQUENCE CHARTS BENEFITS

Message Sequence Charts are quite similar to UML Sequence Diagrams. Roughly speaking, they represent the interaction between a set of agents called instances by means of sending and receiving signals. More information about MSC can be found in ITU-T specifications [3]. Fig. 2 demonstrates an MSC for a part of SIP protocol.

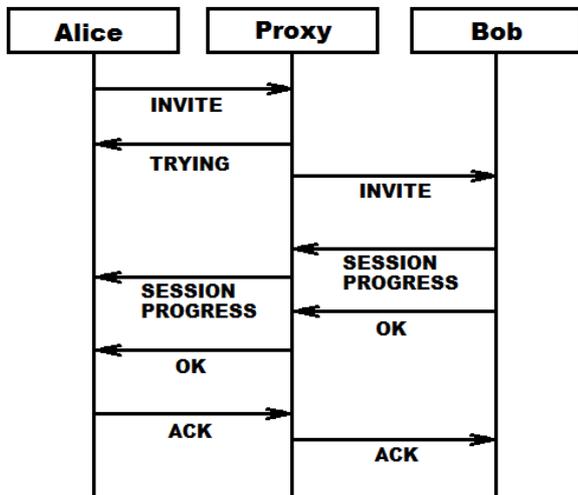


Figure 2. MSC for a part of SIP protocol

It is very native to use sequence diagrams to illustrate use-cases or even provide concrete specification for communication protocols or data exchange between software components. Phrase representation of MSC can be automatically parsed and analyzed. It means that diagrams may be used as test scenarios. In this context, the idea of signal in MSC is very close to the concept of keyword.

MSC as a format for keyword-driven tests has particular advantages. It is human-oriented and can be presented in graphical view [5]. Due to standardization it is possible to develop formal algorithms for MSC processing. Their implementation can rely on third-party tools and libraries. Some of the existing modeling and developer tools provide MSC data, which can be used both for documenting and testing the product.

IV. TECHNOLOGY CONCEPT

Main elements of MSC diagrams are signals, actions and inline expressions. All of them can be used to implement keyword-driven testing concepts.

Signal exchange can be interpreted as a series of keywords, executed one after another. Testing data can be defined in signal parameters. In context of testing all instances present in sequence diagram refer to environment or SUT. Thus, signals sent from environment to SUT describe testing actions, while signals coming from SUT instances specify the reaction to the stimulus. In keyword-driven testing there is no difference between executed commands in scenario. The concept of sending and receiving signals has to be adopted in a way that makes tests more readable whilst clear and unambiguous.

Taking this into account we can consider signals sent from the environment as a pure keyword execution. Signals sent from SUT also have this meaning but should imply additional checks that affect the result of the test. In can be the interpretation of return code, for example. Another approach is to detect special states and events during testing and store them somewhere for future processing by means of signals sent from SUT. For automated keyword testing let's call it "passive driver".

Actions in MSC are used to describe internal events in instances such as data evaluation, or comments. As an element of testing scenario action can be used to increase readability of the test commenting what is happening inside the "black box". In manual testing it can give additional instructions of provide external references to documents. In automated testing actions can serve as a placeholder for executable code for test customization. In both cases actions can contain keywords.

Summing up what has been said it is possible to use MSC for keyword-driven testing and use diagram components to make tests more readable and vivid. But the role of each element must be clearly defined to keep scenarios easy to understand and provide unambiguous way of their creation.

MSC inline expressions allow specifying non-linear control flow. They describe repetitive actions (loops), optional or alternative behavior (opt and alt expression). Combined with condition events (as in Fig. 3), inline expressions can be used to represent if-else and loop statements in their traditional meaning. To enhance the conditions and data manipulations, variables can be declared using MSC text elements and then used in control flow management and testing data.

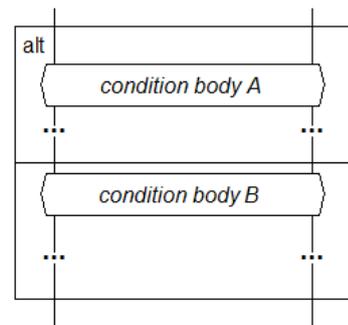


Figure 3. Conditional inline expression

Current research concentrates on automated testing rather than manual. It was mentioned earlier that MSC is standardized notation so it is possible to automatically produce human-readable tests based on it. Testing system can automatically run those tests by leveraging different underlying technologies with driver libraries. During the current research a testing automation framework was created according to the ideas presented above. Main concepts of the framework implementation are presented below with the information about current status of the research and plans for future development.

V. AUTOMATED TESTING FRAMEWORK IMPLEMENTATION

Main idea of the architecture corresponds to the concept of KDT approach. Keywords are extracted from MSC and processed using driver libraries. But instead of direct interpretation of commands tests are first translated to the program in target code. This approach is commonly used in testing with MSC [6]. Then it is build into executable called test unit. Its implementation is based on state machine approach which allows non-linear behavior and automatic analysis of execution trace. Main components of created framework are present in Fig. 4.

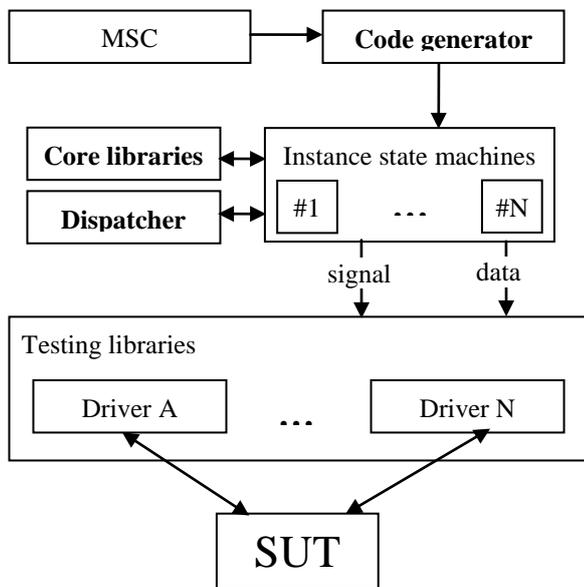


Figure 4. MSC-based testing automation framework

Each MSC environment instance is executed in separate thread. This requires introducing two types of variables – local and shared. Local variables may have different values in different instance threads. To provide thread-safe access to the set of shared variables called test context each test unit has special control module [7]. The latter one is also responsible for processing conditions for inline expressions with multiple instances involved. It tells which branch of alt should be used or whether instances need to execute another iteration of loop.

VI. CONCLUSION

This research is still in progress. Current implementation of framework include translator from MSC to C, core and driver libraries, test report generator. Proposed testing approach is now being approbated in testing of telecommunication software. Future plans include design and implementation of passive driver approach and integration with Robot framework testing libraries.

REFERENCES

- [1] K. Beck, Test-Driven Development by Example, Saint Petersburg: Piter, 2003.
- [2] Faight, Danny R. Keyword-Driven Testing. Sticky Minds. Software Quality Engineering. <http://www.stickyminds.com/article/keyword-driven-testing>, 2004.
- [3] ITU-T Recommendation Z.120: Message sequence chart (MSC). Geneva, Switzerland, October 1996, <http://eu.sabotage.org/www/ITU/Z/Z0120e.pdf>.
- [4] Tiutin B., Veselov A., Kotlyarov V. Scaling of automated test execution // St. Petersburg State Polytechnical University Journal. № 3 (174). 2013. P. 118-122.
- [5] W. Damm, D. Harel. LSCs: Breathing life into message sequence charts. Formal Methods in System Design, 19(1), 2001.
- [6] Hu W., Sun X. Test Case Generation Based on MSC TTCN-3 // Proceedings of the International Conference on Information Engineering and Applications (IEA). London: Springer-Verlag, 2013. 888 p.
- [7] Kaner C., Bach J., Pettichord B. Lessons Learned in Software Testing: A Context-Driven Approach. NY: John Wiley & Sons, Inc., 2001. 320 p.

Reconciliation Testing Aspects of Trading Systems Software Failures

Anna-Maria Kriger
Kostroma State Technological University
anna-maria.kriger@exactpro.com

Vladislav Isaev
Yuri Gagarin State Technical University of Saratov
vladislav.isayev@exactpro.com

Alyona Pochukalina
Obninsk Institute for Nuclear Power Engineering
alyona.pochukalina@exactpro.com

Abstract - This paper describes the concept of reconciliation testing - a process of using data reconciliation tools to validate the system in parallel with other activities. The authors studied information about two major software failures in electronic trading area: Facebook IPO on NASDAQ and Knight Capital runaway algorithms. This paper contributes to the subject matter by identifying aspects related to data reconciliation during these two events. The authors discuss the balance between automated and manual reactions to discrepancies reported by reconciliation tools and analyze the necessity of introducing reconciliation testing as part of system development life cycle for complex transactional processing systems.

Keywords - data reconciliation, software testing, electronic trading

I. Introduction

Reconciliation is a process of finding discrepancies in data obtained from different sources. In accounting, reconciliation refers to the process of ensuring that two sets of records, usually account balances, match to each other. In financial markets, data reconciliation systems help asset managers to reconcile trades, cash and security flows, balances and positions between different systems, e.g. internal data stored by the trading participant vs. external data received from counterparties, brokers, clearers, custodians, etc. [1]. Data reconciliation packages are often used by middle- and back-office teams to identify breaks in post-trade data stored in relational databases. Most of data reconciliation research is also focused on various database related techniques [2]. General purpose extract, transform, load (ETL) products such as Informatica PowerCenter can be used as the basis for reconciliation tools implementations [3]. The financial services industry also uses specialized solutions such as UnaVista [4] from the London Stock Exchange. Data reconciliation can be implemented as:

- a) End of day process
- b) Periodic process
- c) Real-time process

The optimal implementation approach depends on balance between time exposure risks of less frequent solutions and footprint requirements of more frequent solutions. Slower solutions delay the delivery of critical information to the operational team, but require less hardware resources compared to faster solutions. Due to latency requirements, relational databases were removed from the main transactional

path in most of the trading systems [5]. This fact, along with the desire to limit time exposure, is likely to be reflected in the next generation of reconciliation products that will move away from databases and focus more on real-time matching.

In the next part of the paper, Reconciliation Testing concept is described. Parts III and IV cover two samples of major software malfunctions in the electronic trading area. The first one describes events related to a broken Knight Capital algo that submitted millions of uncontrolled orders into the US markets and acquired a huge loss position. The second one describes problems with determining the uncrossing price and sending confirmations to members during Facebook IPO on NASDAQ exchange. The last part contains the analysis of similarities between Knight Capital and Facebook IPO events from data reconciliation and testing points of view.

II. Reconciliation Testing

Reconciliation testing is a process of using data reconciliation tools to validate the system in parallel with other testing activities. The term rarely appears in research papers. Data reconciliation tools can be viewed as passive testing tools due to their ability to check data consistency across the system without initiating any additional message flows [6]. The ability of data reconciliation tools to report errors in data consistency makes them useful test oracles for both functional and non-functional testing activities.

By their very nature, production data reconciliation tools satisfy the requirements for test tools that can be used in trading systems production environments [7]. Thus, data reconciliation tools can support the requirements of High Volume of Test Automation (HiVAT) methods:

- their impact on the system under test is acceptable for both production and test environments;
- tools can collect and process data regarding events in the system under test at production rates/volumes;
- they can highlight discrepancies in large data sets in a form that can be analyzed by the operational or QA teams;
- tools stability and resilience are sufficient to run high volumes of automated tests.

It is important to use data reconciliation tools during negative tests execution. The Quality Assurance team should check whether negative scenarios can be picked by the data reconciliation tools or not. Whenever a negative test scenario leads to a discrepancy highlighted by the data reconciliation tool, the Quality Assurance team should validate whether it is

possible to use the information from the tool to identify the source of the problem. This way, the operational team will have the necessary insight to take action if the problem ever occurs in production environment.

Reconciliation testing requires the presence of data reconciliation tools in the test environments. In some cases it can lead to an additional license costs and other expenses. Yet, the absence of production-like instrumentation limits the coverage of operational testing.

In order to perform reconciliation tests one needs to have data reconciliation tools available. The Quality Assurance team should strongly consider the possibility of implementing test tools capable of running passive data consistency checks. These tools should be implemented with a potential opportunity in mind that they will be also used in production environments.

In summary, the main aspects of data reconciliation tools are:

- a) they are passive test tools
- b) they serve as test oracles
- c) they can be used with HiVAT methods
- d) they should be used during negative test cycles

III. Knight Capital

Knight Capital was one of the most successful high-frequency trading (HFT) companies and represented approximately 10% of listed US equity securities in 2011-2012. Knight operated ultra-fast order router software named SMARS. A technical glitch in the system that happened on the 1st August 2012 led to an uncontrolled order submission into the market and accumulated a loss position of \$460 million within 45 minutes period [8].

Smart Order Router (SOR) software is intended to execute orders in the current fragmented financial markets landscape [9]. Figure 1 below shows a simplified view of SOR system architecture.

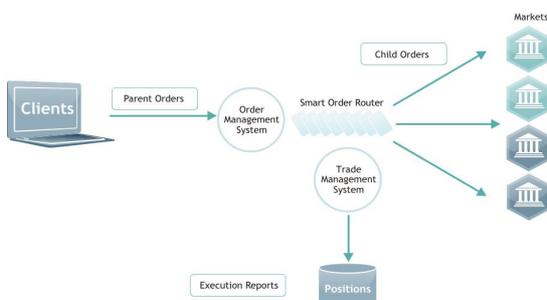


Fig. 1. Simplified SOR architecture

The Order Management System (OMS) receives orders from clients (parent orders) and after validation checks and controls passes them to the SOR subsystem. The latter creates market orders (child orders) for every parent order and sends them to different exchanges depending on the state of the markets and the internal business logic of the system. The information about trades is stored into Trade Management System (TMS) and trading positions and accounts are updated.

A set of reconciliation controls is necessary to protect the system as shown in Figure 2.

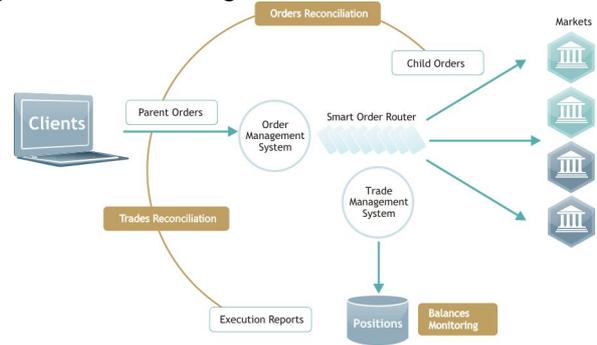


Fig. 2. Reconciliation controls in a SOR system

It is necessary to check the discrepancies between parent and child orders. When child orders are executed in the market it is necessary to reconcile market execution reports vs. parent orders. Whenever discrepancies are detected, they should be reflected in the error accounts.

The SMARS system contained the necessary reconciliation controls. However it appeared that they were not properly configured or tested. Reconciliation control to validate parent orders vs. child orders appeared to be higher in the source code and the SMARS system's logic was not affected by the check. All other risk controls were located at the OMS level and were not suitable to block problems should they happen in the SMARS system. Knight Capital had a special monitoring system called PMON to view positions accumulated in the error account, but its output was not linked to any kill-switch mechanism and did not provide sufficient information to operational teams to understand the source of the problem.

Knight Capital implemented changes related to New York Stock Exchange (NYSE) Retail Liquidity Platform in the SMARS system and put them live on the date of the events. Due to a human operator's error, the changes were deployed on seven servers instead of eight. The newly introduced switch triggered a piece of legacy code on that server, and the result was an uncontrolled flow of child orders into the market. The system continued to send child orders even though client parent orders were already filled. Broken real-time reconciliation controls were not able to halt erroneous run-away trading algorithm and post-trade controls were not designed to affect real-time flow. The Securities and Exchange Commission (SEC) executive order highlighted the lack of technical supervision in the firm and issued additional fine of \$12 million. Knight Capital was not able to recover from these events, its share price dropped and the company was later acquired by one of its competitors.

IV. Facebook IPO

Facebook is the most widely used social network in the world. Its audience grew substantially over the years and exceeded one billion users. In 2012 company announced that it selected NASDAQ market as its listing exchange. Facebook Initial Public Offering (IPO) was one of the largest IPOs in history. Many retail and institutional investors were going to

participate and acquire company shares. On the day of IPO, 18th May 2012, the trading activities in the stock were disrupted by a set of technical malfunctions that lasted for several hours, had substantial financial impact on some of the market participants and led to SEC investigation [10].

The NASDAQ system is one of the most advanced trading platforms used by many national and alternative exchanges in many countries. The system has resilient and scalable distributed architecture and a set of built-in reconciliation controls targeted to validate internal data consistency. The trading system can operate in two different modes – continuous trading and auctions. Continuous trading is a very efficient way to organize markets with sufficient liquidity. Whenever a price of the buy order exceeds or equals the price of the sell order, a trade will happen during continuous trading. Market participants have immediate access to price discovery and trading opportunities. Continuous trading is a self-maintained process. However, is it not the most effective way of starting a new trading day or maintain an orderly market after significant material events. The reason for that is that every participant is afraid that others have information that is not reflected in the share price, as trading had not started yet and thus waits for others to submit their orders. Collectively, this behavior results in limited available liquidity. The problem can be resolved by the auction trading mode. For a designated period of time, the participants can submit, amend and cancel their orders, they can also view prices offered by other participants, but no trades will happen until a particular moment. Auction trading mode gives investor sufficient confidence to decide whether they really have an intention to trade at the price accepted by the market. At the end of the auction call period the exchange system identifies uncrossing price that will result in maximum volume of traded shares and the trading goes into the continuous mode [11]. Secondary trading in the NASDAQ markets usually starts with a special auction called “Display Only Period” (DOP). NASDAQ uses a separate component called “IPO Cross Application”. It processes all orders to define the price at which the largest number of shares will trade and then the matching engine crosses eligible buy and sell orders at that price.

The NASDAQ system has a reconciliation control to validate that the list of orders presented in the matching engine is identical to the one used by the Cross Application to determine the price. This control directly affects the trading system and results in a request to recalculate the price whenever any discrepancy is located. One of the reasons for the reconciliation check to fail was that NASDAQ allowed participants to cancel orders even during a short period of uncrossing price calculation that usually takes 1-2 milliseconds.

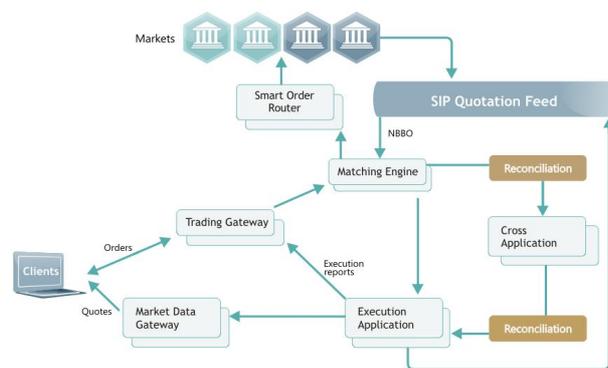


Fig. 3. Conceptual exchange trading system architecture

Information about the NASDAQ platform components and its architecture is not available in the public domain. We presented a generic view of the simplified architecture in Figure 3 based on information from the SEC report and our overall experience with similar systems. Clients submit orders into the trading gateways and orders are matched inside the matching engine. According to the report, the system uses separate components to calculate the uncross price during auctions and another application to send confirmation reports to members and publish quote updates called Execution Application. Similar to the Cross Application, the Execution Application also had associated reconciliation controls in place to make sure that its view of the orders match to the one available in the Cross Application.

On the day of Facebook IPO, the NASDAQ platform received unusually high number of orders from participants desiring to participate in market opening auction. The IPO Cross Application process took around 20ms to determine the uncrossing price and a single order was canceled during this period. The application repeated the calculation and reconciliation check, but more orders were canceled. The NASDAQ matching engine and the IPO Cross Application went into an infinite loop. Every attempt to recalculate the uncrossing price was followed by failed reconciliation check. Within the next 25 minutes, technical and executive teams determined that the reconciliation check prevented the system from opening the market and agreed on a so called Failover Proposal. Software update switching of the check was deployed on the secondary server and the primary one was killed, enabling the system to stop the cycle. Unknown at the time, due to the ongoing cycle the system’s ability to process additional inbound order instructions was limited, and an extra 38k orders were stuck in the processing queue and did not participate in the cross. This led to the failure of reconciliation check in the Execution Application. Many market participants were not able to receive confirmations for their orders and trades until NASDAQ performed the second failover and switched off the reconciliation control in the Execution Application. Figure 4 shows the state of the system after both failover proposals were executed.

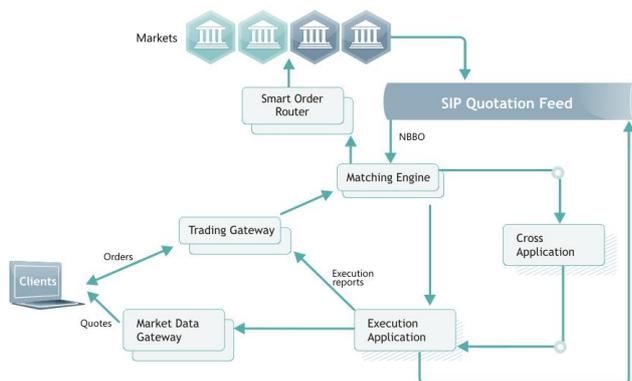


Fig. 4. NASDAQ system state after executing the second failover proposal

The events around Facebook IPO resulted in significant loss of investors' confidence, the NASDAQ operator was censured by SEC and had to pay an administrative penalty of \$10 million and set aside a \$62 million-worth fund to compensate firms harmed by the glitch.

V. Reconciliation Testing Analysis

Large-scale technology disasters are rarely a consequence of a single factor. Mostly, they result from a set of flaws in software development and maintenance processes. Data reconciliation controls serve as an independent additional protection mechanism for complex systems and therefore should be considered as a necessary part of production infrastructures. Reconciliation testing is an activity that not only helps to deliver systems that will behave correctly in production, but also provides additional confidence that operational teams will have sufficient information to take action if things unexpectedly go wrong.

In both the Knight Capital and the Facebook IPO cases, the trading systems had a reasonable set of reconciliation controls. In both cases, the impact of problems might have been significantly reduced had these controls worked properly. This section covers distinctions and similarities between the two considered events.

The correctness of real-time reconciliation control matching parent orders vs. child orders had not been tested by Knight Capital for several years. A negative scenario that resulted in a discrepancy between these two data sets could have highlighted that the risk control was longer active after being moved into another part of the source code. On the other hand, it is clear that reconciliation controls had been functionally tested by NASDAQ and proved to work as expected. However, the exchange team had never tested the course of actions if the reconciliation control failed permanently. The team executed the failover proposal without validating in detail first what impact it would have on other components and reconciliation utilities.

Both companies had a monitoring view that highlighted the problem to their operational teams. In both cases, the team was able to correctly interpret the extent of the events. The Knight Capital team erroneously decided to roll-back the changes and effectively made the things worse. The NASDAQ

team was not aware of 38k orders stuck in the processing queue for some time, even though the reconciliation control in the Execution Application had picked up the problem immediately and marked the cross as invalid.

The Knight Capital reconciliation tools were not linked to any facilities to halt the trading. In the NASDAQ case, failed reconciliation immediately blocked further processing. Upon reflection, it is clear that neither of these two behaviors is ideal. It is necessary to have balance between automated stop-switches and the operators' ability to control reconciliation checks.

In both cases, real-time data reconciliation controls were built into the main transactional part. It might be a good idea to use tools separated from the main flow, e.g. surveillance sub-systems, to perform the data reconciliation function.

The following figure shows market surveillance system usages as the test tool.

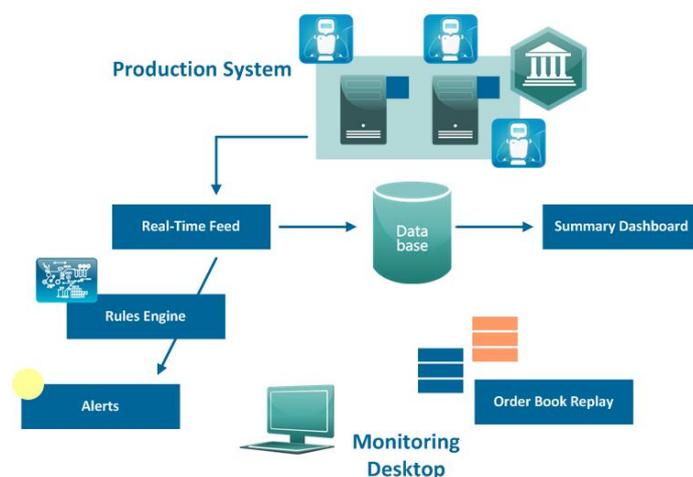


Fig. 5. Market Surveillance System used as reconciliation testing tool.

The primary task of a market surveillance system is to support the analytics gathered and analyzed by departments responsible for recognition of possible market abuse [12]. A surveillance system must collect the information pertaining to all incoming orders, system responses, data from external sources and relevant internal states of the trading platform.

It is possible and beneficial to use market surveillance system as a reconciliation testing tool for the following reasons:

- all required data is collected from the system and available both real-time and in the database;
- most of surveillance systems are configured as a downstream component and do not affect the main transactional path;
- rules engine allows creating data reconciliation checks and raise alerts when they fail;
- order book replay allows studying the exact source of the discrepancy.

VI. Conclusion

The examples of high-profile software failures presented in the paper show that incorrectly functioning data reconciliation controls in electronic trading systems can cause substantial financial losses. Validation of these controls needs to be incorporated into the software development life cycle for such systems.

A comprehensive test library should cover various potential discrepancies reported by data reconciliation tools. Operational teams should provide responses to each of these scenarios. Quality Assurance teams should verify that the tools provide enough information to identify the source of a discrepancy. The system itself should have controls to halt and resume trading both automatically and manually if a breakdown occurs in production environment.

Apart from being a critical part of production infrastructures, data reconciliation tools can provide additional test oracles for both functional and non-functional testing activities and enable more efficient testing of complex transactional processing systems.

The authors plan to proceed with researching data reconciliation tools applicability in software testing and developing a reference implementation of a scalable real-time tool for reconciliation testing based on the proprietary market surveillance platform.

References

- [1] W. Wheatley Financial Systems, *Reconciliation Best Practice*, <http://www.watsonwheatley.com/literature.html>
- [2] M. Cochinwala, V. Kurien, G. Lalk, D. Shasha, "Efficient data reconciliation", *The Journal of Information Science*, vol.137, issue 1-4, Sep. 2001
- [3] R. Nolan, The Informatica Blog, *Even 'The Most Interesting Man In The World' Won't Do This...* <http://blogs.informatica.com/perspectives/2012/03/06/even-the-most-interesting-man-in-the-world-wont-do-this/>
- [4] London Stock Exchange. *How UnaVista Works*: <http://www.londonstockexchange.com/products-and-services/matching-reconciliation/how-unavista-works/index.html>
- [5] I. Itkin, *Highload trading systems and their testing*, Highload++ 2012
- [6] A. Matveeva, N. Antonov, I. Itkin, "*The Specifics of Test Tools Used in Trading Systems Production Environments*", Tools & Methods of Program Analysis 2013
- [7] A. Alexeenko, P. Protsenko, A. Matveeva, I. Itkin, D. Sharov, "*Compatibility Testing of Protocol Connections of Exchange and Broker Systems Clients*", Tools & Methods of Program Analysis 2013
- [8] SEC Release No. 70694. *In the Matter of Knight Capital Americas LLC*
- [9] Foresight: *The Future of Computer Trading in Financial Markets* (2012) Final Project Report
- [10] SEC Release No. 69655. *In the Matter of THE NASDAQ STOCK MARKET, LLC*
- [11] *NASDAQ Stock Market Rules* <http://nasdaq.cchwallstreet.com/>
- [12] D. Diaz, M. Zaki, B. Theodoulidis, P. Sampaio, *A Systematic Framework for the Analysis and Development of Financial Market Monitoring Systems*, Annual SRII Global Conference 2011

Simulation-based Hardware Verification Back-end: Diagnostics

Mikhail Chupilko, Alexander Protsenko
Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
{chupilko,protsenko}@ispras.ru

Abstract—Hardware development processes include verification as one of the most important part. Verification is very often done in simulation-based way. After comparison of design and its reference model behavior, the verdict about their correspondence appears. It is very useful to have some means of analyzing potential inconsistency of their output data. It is exactly the subject of this work to supply verification engineers with a method and a back-end tool for diagnostics of incorrect behavior using wave diagrams and reaction trace analysis based on recombination of reaction traces.

I. INTRODUCTION

The importance of hardware verification (taking up to 80% of the total development efforts [1]) is raised by difficulties in error correction in already manufactured devices. Many methods address verification, some of them being more formal (*static analysis*), the other ones using simulators (*dynamic verification*). In the first case, the verification is carried out in a strict mathematical way. For example, the approach to verification known as *model checking* [2] means checking satisfiability of formally expressed specification and formulae manually created by an engineer. If some error occurs, it is connected with unsatisfiability of the properties set by the engineer and the specification that should be checked and corrected. Dynamic verification implies checking mutual correspondence of output reactions of two models: *design under verification (DUV)* and a *reference model* (possibly, expressed by a set of *assertions*). The same stimulus sequence is applied to the both models, their reactions are checked and if some problem occurs, incorrect reactions are shown (the reactions can be so due to their time, data, and possibility of their appearing). As the unarmed looking at incorrect reactions is not always enough to understand quickly the problem having occurred, it seems very important to show more diagnostics information, including the place of this error on the wave diagram, and the answer to the question why such an error has appeared.

In this paper, we will introduce the way of diagnostics, which should be independent of the specification organization. It should be also supported by a wide range of test system making technologies, including the one made by our team (*C++TESK Testing ToolKit* [3], [4]) and widely distributed world-known methods of test system development (*Universal Verification Methodology* [5]).

This work evolves the research previously described in [6] and extends it by new understanding of the explanatory rules

used in the analysis algorithm and visualization of diagnostics.

The rest of the paper is organized as follows. The second section is devoted to related works on the subject of diagnostics and trace analysis. The third section introduces architecture of test systems for simulation-based verification and the proposed method of *diagnostics subsystem* construction. The fourth section considers method implementation and examples of its work with test system development library C++TESK Testing ToolKit. The fifth section concludes the paper.

II. RELATED WORKS

The problem of diagnostics of event-based systems is studied under different angles. Some researchers understand failure diagnosis as checking of formal properties in formally expressed systems (e.g., [7]). In the other sources the fault diagnostics is closer to our task where it means construction of such *timed automata*, which can find a bug in behavior of DUV according to some pattern during the simulation (e.g., [8] and [9]).

The processing of reaction traces produced by DUV and the reference model can be also called *trace rewriting* (e.g., [10]). This term is used for describing of *symbolic trace* reducing methods based on some set of rules. There are several types of objects, which can be removed from the trace without losing of the trace expressiveness, including extra data, dependent objects, and all the others not influencing the analyzed result. In our case, the reaction trace can be represented as a symbolic trace leading to an error occurred at some moment of time. Having information about parent-child dependences between stimuli and reactions, we can remove unnecessary objects from the trace and provide verification engineer with a meaningful essence for analysis of defect.

The task we formulated for the research is quite close in general sense to trace rewriting but has some technical differences including usage of wave diagrams for visualization of diagnostics results, different set of rules accounting peculiarities of HDL design traces, for example *signal interfaces* (sets of HDL signals) where reactions appear.

III. DIAGNOSTICS SUBSYSTEM

The being developed diagnostics subsystem should be a back-end to common simulation-based test system architecture. To understand the position of the back-end better, let us review quite common architecture with names of objects from C++TESK (see Figure 1). The concrete architecture selection

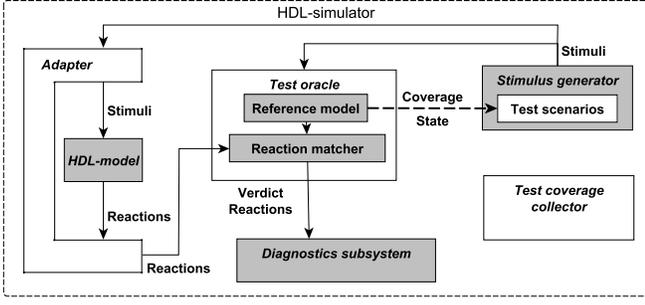


Fig. 1. Common architecture of test system

does not mean a lot as the set of objects in mentioned above wide-distributed UVM is very similar to those in C++TESK ([11]).

The typical components of simulation-based test systems are *stimulus generator*, *test oracle* (including *reaction matcher*), *adapter* making the interface between transaction level test system and signal-level DUV, *test coverage collector*. The diagnostics subsystem is also shown in the Figure 1 to clarify its position. After reaction matcher having checked correspondence of two reaction traces, the one from DUV, another from the reference model and produced the false verdict at the current cycle, all this information is given to the diagnostics subsystem, which can work as an extern plug-in for the test system.

The input data for the diagnostics subsystem is a trace including reactions from reference model and DUV, applied stimuli, dependences between them (by some *parent identifier*). All this objects can be provided in XML. The diagnostics subsystem can also process wave diagrams to show the important signals and position of defect reactions. The latter also requires *mapping of DUV signals* to reactions in XML.

Let the reaction checker use two sets of reactions: $R_{spec} = \{r_{spec_i}\}_{i=0}^N$ and $R_{impl} = \{r_{impl_j}\}_{j=0}^M$. Each specification reaction consists of four elements: $r_{spec} = (data, iface, time_{min}, time_{max})$. Each implementation reaction includes only three elements: $r_{impl} = (data, iface, time)$. Notice that $time_{min}$ and $time_{max}$ show an interval where specification reaction is valid, while $time$ corresponds to a single time mark: generation of implementation reaction always has concrete time.

The reaction checker has already attempted to match each reaction from R_{spec} with a reaction from R_{impl} , having produced *reaction pairs*. If there is no correspondent reaction for either specification or implementation ones, the reaction checker produces some pseudo reaction pair with the only one reaction. Each reaction pair is assigned with a certain type of situation from the list of *normal*, *missing*, *unexpected*, and *incorrect*.

For given reactions $r_{spec} \in R_{spec}$ and $r_{impl} \in R_{impl}$, these types can be described as in Table I. Remember that each reaction can be simultaneously located only in one pair.

The diagnostics subsystem has its own simplified inter-

Type name	Reaction pair	Definition of type
NORMAL	(r_{spec}, r_{impl})	$data_{spec} = data_{impl}$ $iface_{spec} = iface_{impl}$ $time_{min} < time < time_{max}$
INCORRECT	(r_{spec}, r_{impl})	$data_{spec} \neq data_{impl}$ $iface_{spec} = iface_{impl}$ $time_{min} < time < time_{max}$
MISSING	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \setminus R_{impl}^{normal, incorrect} : iface_{spec} = iface_{impl} \text{ \& } time_{min} < time < time_{max}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R_{spec}^{normal, incorrect} : iface_{spec} = iface_{impl} \text{ \& } time_{min} < time < time_{max}$

TABLE I
REACTION CHECKER REACTION PAIR TYPES

Type name	Reaction pair	Definition of type
NORMAL	(r_{spec}, r_{impl})	$data_{spec} = data_{impl}$
INCORRECT	(r_{spec}, r_{impl})	$data_{spec} \neq data_{impl}$
MISSING	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \setminus R_{impl}^{normal, incorrect}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R_{spec}^{normal, incorrect}$

TABLE II
DIAGNOSTICS SYSTEM REACTION PAIR TYPES

pretation of reaction pair types (see Table II). In fact, the subsystem translates original reaction pairs received from the reaction checker into the new representation. This process can be described as $M \Rightarrow M^*$, where $M = \{(r_{spec}, r_{impl}, type)_i\}$ is a set of reaction pairs marked with *type* from the list above. $M^* = \{(r_{spec}, r_{impl}, type^*)_i\}$ is a similar set of reactions pairs but with different label system. It should be noticed that there might be different M^* according to the algorithm of its creation (accounting for original order, strategy of reaction pair selection for recombination, etc.).

To make the translation, the diagnostics subsystem uses a set of reaction trace *transformation rules*. Each of the rules transforms the reaction pairs from the trace but does not change their data. To find the best rule for application, the subsystem uses a *distant function*, showing the closest reactions among the pairs. The distant function can be implemented in three possible ways.

Metric 1: Reaction closeness correlates with the number of equal data fields of two given reactions.

Metric 2: Reaction closeness correlates with the number of equal bits in data fields of two given reactions (the Hamming distance).

Metric 3: Reaction closeness correlates with the number of equal bits in data fields of two given reactions, order of equal and unequal areas, and their mutual disposition.

The measure of closeness between two given reactions is denoted as $c(r_{spec}, r_{impl})$.

Each rule processes one or several reaction pairs. In case of missing reaction or unexpected reaction, one of the pair elements is undefined and denoted as *null*. Each reaction pair

is assigned with a signal interface. The left part of the rule shows initial state; the right part (after the arrow) shows result of the rule application. If the rule is applied to several reaction pairs, they are separated with commas.

In general, the algorithm of rule application consists of two stages. At the first stage, reactions with equal data are joined and transformed by the rules. In case of such a transformation, the application order of rules is of importance. The second stage includes processing of the rest reactions and new ones made at the first stage. Here rule priority is less important than values of the selected distant function.

Now, let us review all the six rules that we found including the first two rules being basic. In description of the rules c means the selected distant function but at the first stage of the algorithm it is always full equivalence of data. At the second stage of the algorithm a rule may be applied only if c value for this rule is the best among c values for the other rules for given reactions.

Rule 1: If there is a pair of *collapsed reactions*, it should be removed from the list of reaction pairs. $(null, null) \Rightarrow \emptyset$.

Rule 2: If there is a normal reaction pair $(a_{spec}, a_{impl}) : data_{a_{spec}} = data_{a_{impl}}$, it should be *collapsed*. $(a_{spec}, a_{impl}) \Rightarrow (null, null)$.

Rule 3: If there are two incorrect reaction pairs with mutual correlation of data, the reaction pairs should be regrouped. $\{(a_{spec}, b_{impl}), (b_{spec}, a_{impl})\} : c(a_{spec}, a_{impl}) < c(a_{spec}, b_{impl}) \ \& \ c(a_{spec}, a_{impl}) < c(b_{spec}, a_{impl})$ or $c(b_{spec}, b_{impl}) < c(a_{spec}, b_{impl}) \ \& \ c(b_{spec}, b_{impl}) < c(b_{spec}, a_{impl})$ (this closeness is the best among the other rules), $\{(a_{spec}, b_{impl}), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, b_{impl})\}$

Rule 4: If there is a missing reaction pair and an unexpected reaction pair with mutual correlation of data, these reaction pairs should be united into one reaction pair. $(a_{spec}, null), (null, a_{impl})$ and $c(a_{spec}, a_{impl})$ is the best among the other rules: $\{(a_{spec}, null), (null, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl})\}$.

Rule 5: If there is a missing reaction pair and an incorrect reaction pair with mutual correlation of data, these reaction pairs should be regrouped. $(a_{spec}, null), (b_{spec}, a_{impl})$ and $c(a_{spec}, a_{impl}) < c(b_{spec}, a_{impl})$ (this closeness is the best among the other rules), $\{(a_{spec}, null), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, null)\}$.

Rule 6: If there is an unexpected reaction pair and an incorrect reaction pair with mutual correlation of data, these reaction pairs should be regrouped. $(null, a_{impl}), (a_{spec}, b_{impl})$ and $c(a_{spec}, a_{impl}) < c(a_{spec}, b_{impl})$ (this closeness is the best among the other rules), $\{(null, a_{impl}), (a_{spec}, b_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (null, b_{impl})\}$.

The first stage of the algorithm is shown in 1 and 3 action blocks. The first stage having passed, the sets R_{spec} and R_{impl} does not contain any not yet collapsed reactions with identical data. The time of the second stage comes (see action blocks 2 and 4). Both stages of the algorithm having passed, the list of reaction pairs may contain some reaction pairs with both specification and implementation parts but not collapsed due

to their unequal data. To show diagnostics info for them too, they are collapsed using modified second rule, not requiring equality of data in the reaction pairs.

After the application of each rule, the history of transformation is traced and then it is possible to reconstruct the predecessors of the given reaction pairs and all the rules they were processed by. Such a reconstruction of the rule application trace we understand as the *diagnostics information*.

In the result, verification engineers are provided with a list of problems occurred during verification and with a set of hints making bug localization easier.

Action 1 $match1[(r_{1_{spec}}, r_{1_{impl}}), (r_{2_{spec}}, r_{2_{impl}}), r_{numb}]$

Input: $RP_1 = (r_{1_{spec}}, r_{1_{impl}}), RP_2 = (r_{2_{spec}}, r_{2_{impl}}), c = total_equivalence$
for all $rule \in |Rules|$ **do**
 if $rule.isApplicable(RP_1, RP_2, c)$ **then**
 $r_{numb} \leftarrow rule.number$
 return true
 end if
end for
return false

Action 2 $match2[(r_{1_{spec}}, r_{1_{impl}}), (r_{2_{spec}}, r_{2_{impl}}), r_{numb}]$

Input: $RP_1 = (r_{1_{spec}}, r_{1_{impl}}), RP_2 = (r_{2_{spec}}, r_{2_{impl}}), c = selected_metric_function$
 $metric \leftarrow 0$
for all $rule \in |Rules|$ **do**
 $metric^* \leftarrow rule.getMetric(RP_1, RP_2, c)$
 if $(metric^* > metric)$ **then**
 $metric \leftarrow metric^*$
 $r_{numb} \leftarrow rule.number$
 end if
end for
return metric

Action 3 $apply_stage1[\{(r_{spec}, r_{impl})_i\}]$

Input: $\{(r_{spec}, r_{impl})_i\}$
for all $r \in |\{(r_{spec}, r_{impl})_i\}| \ \& \ !r.collapsed$ **do**
 for all $p \in |\{(r_{spec}, r_{impl})_i\}| \ \& \ !p.collapsed$ **do**
 if $match(r, p, rule_number)$ **then**
 $(r_{spec_{i+1}}, r_{impl_{i+1}}), (r_{spec_{i+2}}, r_{impl_{i+2}}) \leftarrow Rules[rule_number].apply(r, p)$
 $r.collapsed \leftarrow true$
 $p.collapsed \leftarrow true$
 return
 end if
 end for
end for

Action 4 $apply_stage2[\{(r_{spec}, r_{impl})_i\}]$

Input: $\{(r_{spec}, r_{impl})_i\}$

for all $r \in |\{(r_{spec}, r_{impl})_i\}| \&!r.collapsed$ **do**
 $metric^* \leftarrow 0$
for all $p \in |\{(r_{spec}, r_{impl})_i\}| \&!p.collapsed$ **do**
 $metric = fuzzy_match(r, p, rule_number)$
if $metric > metric^*$ **then**
 $metric^* \leftarrow metric$
 $rule_number^* \leftarrow rule_number$
 $s_1 \leftarrow r$
 $s_2 \leftarrow p$
end if
end for
if $metric^* > 0$ **then**
 $(r_{spec_{i+1}}, r_{impl_{i+1}}), (r_{spec_{i+2}}, r_{impl_{i+2}}) \leftarrow$
 $Rules[rule_number^*].apply(s_1, s_2)$
 $s_1.collapsed \leftarrow true$
 $s_2.collapsed \leftarrow true$
return
end if
end for

IV. IMPLEMENTATION OF THE METHOD

The proposed approach to diagnostics of incorrect output reactions has been implemented as a plugin in C++ and Java languages and attached to C++TESK Testing ToolKit [4].

If the verification process fails, the information provided by the diagnostics subsystem is shown. It looks like tables with all found errors (see Figure 2) and rule application history: new reaction pair sets and the way of their obtaining.

Now let us proceed to the following part of diagnostics subsystem work — the visualization of bugs on wave diagrams. Each specification reaction produced during test process keeps its parents — stimuli and other events making this reaction. Therefore, it is possible to reconstruct the whole chain from the very first stimulus up to the reaction with one of the error types. Each reaction contain data that correspond to signals of HDL model. Typically, the HDL signals are grouped into input and output interfaces and correlate with names of data fields in reactions. There should be a map between signals of interfaces and data fields. Such a map is usually created manually before development of test system. Basing on the resulted reaction pairs, a wave diagram produced by simulator (VCD file [12]), and the signal mapping the diagnostics subsystem creates a set of source files for GTKWave [13] to make errors be visual. The diagnostics subsystem creates separated directory with VCD and SAV files for each incorrect reaction pair. According to these files, GTKWave is asked to show the error situation with its history (predecessors), highlighting only those signals which are necessary for understanding the situation. These signals include ones from output interfaces used in reactions and some common signals like clock, reset and so on. It is possible to show the reference values of signals by injecting into VCD files special signals and labeling them as the reference ones for so and so signals. This possibility

Failure #1	IFACE VIOLATION? [iface41]:8 2 time(s)					
OutputData	data0	data1	data2	data4		
Received	b74426de4836da5c	84c630d7ce01f086	1d4cfa86f2955c53	0		
Expected	b74426de4836da5c	84c630d7ce01f086	1d4cfa86f2955c53	0		
Interface	expected on [iface38]					
Statistics	STIMULI	REACTIONS	NORMAL	INCORRECT	MISSING	UNEXPECTED
1.12 (r/s)	8	9	3	0	2+2	2
Simulation	3013 cycle(s) / 139888886 sec(s) / 0.00 Hz					

Fig. 2. Result of the diagnostics subsystem work

has not been implemented yet but there is no technological difficulty as the diagnostics subsystem already parses VCD files and creates new files with subset of signals.

Example of visual representation of the error from Figure 2 is shown in Figure 3. The situation described by these figures is as follows. The reaction expected at the 38th interface was received at the 41st interface. First, it resulted in missing and unexpected reactions, and then the diagnostics subsystem joined these reactions to create a normal one. The situation of the reaction appearing at the 41st interface and the reaction absence at the 38th interface is exactly shown in the Figure 3.

V. CONCLUSION

The proposed means for trace analysis and bug visualization allows in some sense to make the verification easier. It allows to avoid extra information from the reaction trace and to show only meaningful information for verification engineers related to the occurred and examined bug in HDL designs.

Our future research is connected with localization of found problems and bugs in HDL designs using static analysis of source code.

REFERENCES

- [1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Pub, 2003.
- [2] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [3] M. Chupilko and A. Kamkin, "Specification-driven testbench development for synchronous parallel-pipeline designs," in *Proceedings of the 27th NORCHIP*, nov. 2009, pp. 1–4.
- [4] C++tesk homepage. [Online]. Available: <http://forge.ispras.ru/projects/cpptesk-toolkit/>
- [5] Unified verification methodology. [Online]. Available: <http://www.uvmworld.org>
- [6] M. Chupilko and A. Protsenko, "Recognition and explanation of incorrect behaviour in simulation-based hardware verification," in *Proceedings of the 7th SYRCoSE*, 2013, pp. 1–4.
- [7] S. Jiang and R. Kumar, "Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications," in *IEEE Transactions on Automatic Control*, 2001, pp. 128–133.
- [8] S. Tripakis, "Fault diagnosis for timed automata," in *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2, ser. FTRTFT '02*. London, UK, UK: Springer-Verlag, 2002, pp. 205–224. [Online]. Available: <http://dl.acm.org/citation.cfm?id=646847.707114>
- [9] P. Bouyer and F. Chevalier, "Fault diagnosis using timed automata," in *Foundations of Software Science and Computational Structures: 8th International Conference, FOSSACS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*. Springer-Verlag, 2005, pp. 219–233.

From Abstract Parsing to Abstract Translation

Semen Grigorev

St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia
Email: rsdpisuy@gmail.com

Iakov Kirilenko

St. Petersburg State University
198504, Universitetsky prospekt 28
Peterhof, St. Petersburg, Russia
Email: jake@math.spbu.ru

Abstract—String-embedded language transformation is one of the problems which can be faced during database and information system migration. The conventional solution which is provided by a number of tools is based on run-time translation. We present a static *abstract translation* approach which originates from the *abstract parsing* technique [9] initially developed for syntax analysis of string-embedded languages. We present abstract translation algorithm and some optimization techniques, and discuss the results of its evaluation on a real-world industrial application.

I. INTRODUCTION

Complex information systems are often implemented using more than one programming language. Sometimes this variety takes form of one *host* and one or few *string-embedded* languages. Textual representation of clauses in a string-embedded language is built at run time by a host program and then analyzed, compiled or interpreted by a dedicated runtime component (database, web browser etc.) Most general-purpose programming languages may play role of the host; one of the most evident examples of string-embedded language is dynamic SQL which was specified in ISO SQL standard in 1992 [7] and since then is supported by the majority of DBMS.

String-embedded languages may help to compensate the lack of expressivity of general-purpose language in a domain-specific settings or to integrate heterogeneous components of large system; however this approach comes with some price. In particular even the syntax analysis of string-embedded part of a system is undecidable in general case since its source code is represented implicitly using string-manipulation primitives, procedures and libraries, and generated “on the fly”. In a naïve implementation syntax analysis of embedded clauses is completely outsourced to the runtime environment which postpones many errors from being discovered prior to execution and thus compromises the ideas of code safety and static control.

Abstract parsing is the approach which was developed to overcome the aforementioned deficiency. In abstract parsing the source code of a host application is statically analyzed to provide some constructive representation of the set of string-embedded language clauses which can possibly be generated at run time [9], [4]. This representation is then analyzed by a certain parsing algorithm which is usually derived from some existing one for plain strings [5]. Abstract parsing technique is utilized in a number of tools [8], [10], [2], [3] for program analysis and understanding.

While abstract parsing can help in application analysis it cannot handle the case of application *transformation*. As a

practical use case for string-embedded language transformation we can mention reengineering. During reengineering it is sometimes necessary to migrate from one database management system to another; this migration may require a transformation of string-embedded clauses.

One of the options is dynamic translation at run time [1]. However this solution not always desirable. First, it may degrade the performance of the system due to introduction of extra processing stage. Next, with dynamic translation the ultimate goals of the reengineering are not achieved since some part of the original system escaped transformation.

Another approach includes translation of stored SQL which is supported by a number of existing production tools for database application development [11], [13], [12]. However, these tools do not support dynamic SQL translation and thus provide only partial solution.

The contribution of this paper is an approach for *abstract translation*. Similar to abstract parsing first we perform static analysis to build an approximation for the set of all generated clauses. Then our algorithm performs analysis which, unlike abstract parsing, produces *parsing forest* — a family of syntax trees, each of which represents the result of translation of certain input sequence. New correct assignments for all relevant string values in the host program are calculated then. Our approach works only when the source and the target languages are syntactically close enough (e.g. when they are two dialects of the same language). We discuss some heuristic which helps to reduce the complexity of the algorithm in many practical cases and present the results of its application for the migration of a real-world industrial project from MS-SQL Server 2005 to Oracle 11gR2 platform.

II. GRAPH-BASED INPUT REPRESENTATION

As we mentioned above, the first stage of abstract parsing is static approximation of relevant string values. Two main representations for approximated values were used so far. In [4], [2], [3] the sets of potential string values are described using regular expressions; in [9] approximations are represented in more implicit form as a solutions of a system of (recursive) dataflow equations.

We did not find a way to scale either of these representations to abstract translation case. Instead, we represent the input stream for abstract translation via flow graph with one source and one sink nodes and string-labeled edges. The labels of edges in this graph represent the results of constant propagation so that every path in input graph corresponds to

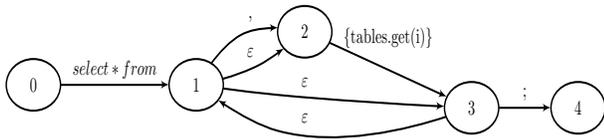


Fig. 1. Correct finite automaton (graph representation)

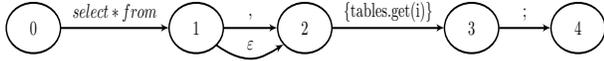


Fig. 2. Result of cycles approximation

one potential value of dynamic query. Moreover, we perform lexical analysis on graphs which converts the initial string-labeled graphs into graphs, labeled by tokens.

Since any cycle in the input graph generates infinite sequence of tokens which upon translation is turned into infinite forest we simplify the graph even more. We replace each cycle with the single repetition. For example in the order to get regular approximation of query value set from code presented below we should build the next regular expression:

`”select * from ” · ({tables.get(i)}|ε)* · ”;`

and the corresponding finite automaton (see Fig. 1). Note that we do not care about approximation for `tables.get(i)` expression because it depends on a constant propagation algorithm. But we will get the graph where cycle replaced with only one repetition of its body. The result of such approximation is presented in Fig. 2.

```
query = "select * from ";
for(int i = 0; i < tables.size(); i++)
{
    if(i != 0) query += ", ";
    query += tables.get(i);
}
query += " ;";
```

As you can see, in our example we do not produce strings like `select * from tbl1, tbl2;` or `select * from ;`. So, we can not check it. We process only two strings and all of them are in original infinite set: `select * from tbl1;` and `select * from, tbl1;`. As a result, we do not process all possible values but we process all variables used for query construction and it is enough for such tasks as code highlighting or transformations because all parts of expression are processed.

Thus the graph becomes cycle-free and we can process all vertices in the topological order. While this drastic simplification is completely heuristic our experience of dynamic SQL translation for real information systems showed that DAG is still a good approximation for practical use.

As an example consider the following code snippet:

```
(1) IF @X = @Y
(2) SET @TABLE = '#tbl1'
(3) ELSE
```

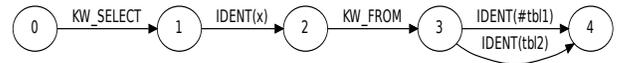


Fig. 3. Tokenized input graph

```
(4) SET @TABLE = 'tbl2'
(5) SET @S = 'SELECT x FROM ' + @TABLE
(6) EXECUTE (@S)
```

Variable `@S` contains dynamically generated query and can have two potential values at the point of query execution. During approximation we can build a graph which represents the set of potential values of the variable `@S` at the line 6. Each edge of this graph is labeled by a token which represents a part of the query (see Fig. 3).

Note that real-world systems can communicate with other systems source of which may be inaccessible to analyze. These systems can contain parts of queries to process and we should use some approximations. For example, clients applications of information system can send conditions for filters (conditions for `where` clause of `select` statement) as part of requests.

III. ABSTRACT TRANSLATION ALGORITHM

Our approach for abstract translation borrows the idea of reusing the control structures used in classical parsing from [9]. Control tables of LALR analyzer may be generated by some conventional tool (e.g. `yacc`¹). The interpreting automaton, however, has then to be modified to be able to compute all possible parser states for each vertex of the input graph.

For example, let we have the following grammar:

```
s -> Ae
e -> BD
e -> CD
```

An input graph is shown on the Fig. 4. The set of parser states for each vertex of the graph can be calculated during syntax analysis. The result of state calculation is shown on the Fig. 5.

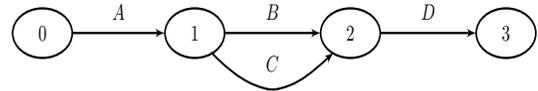


Fig. 4. Input graph for abstract parsing

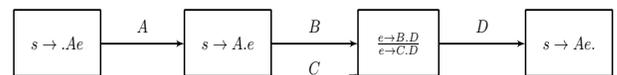


Fig. 5. Parser states

In the case of translation (not parsing) the parsing state consists of state of the automaton and some *semantic* value which represents the result of translation built so far. In

¹<http://dinosaur.compilertools.net>

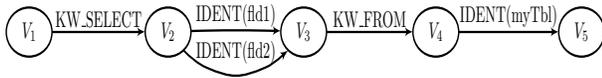


Fig. 6. Graph with states possible to merge

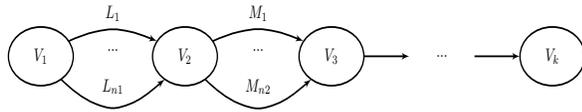


Fig. 7. Graph which requires an exponential resources for translation

particular, the translation algorithm works not only with token types, but also token values.

One of the possible solution of translation is abstract parsing algorithm with mechanism of stack splitting for semantic calculation support. It disallows to merge states and creates a new copy of the whole stack for the each branch of the input graph.

However, this approach faces the exponential memory usage problem. For example parser states for vertex V_3 on the Fig. 6 should be equal for two input edges but if we want to calculate semantics, then we get two different states because identifiers has different values.

Queries which contain a huge number of branches is a big problem. The number of states is an exponential function of the number of branches because for each branch we should produce $n * k$ states where n is a number of states in the root of the fork vertex and k is the number of branches. One of the most frequent example of queries with big number of branches is `select` query. Each of fields to select can be calculated with if-statement or case-statement. Example of such graph is presented on the Fig. 7.

If we use only sequentially concatenated if-statements then the number of parsing trees is 2^n where n is a number of if-statements (or number of branches). In some real-world systems we have faced the queries which contains more than 100 branches. The full forest calculation by naive adaptation of abstract parsing is impossible for such queries.

We propose the following solution for the forest size minimization problem. We have previously mentioned that the result of translation is a new values for all variables which were used for queries construction. It is sufficient to construct not the full forest but only the minimal set of trees such that after translation every variable gets new value. This way, we can process not all paths in the input graph but only minimal set which contains all edges. Note that we cannot calculate this set prior to the parsing because we cannot be sure that every path produces syntactically correct value. If some path contains error than the tree for that path is not constructed and we may lose information about variables. For example consider the graph presented on the Fig. 8.

The one possible set of paths which we can calculate before syntax analysis is $\{(L_1; M_1); (L_2; M_2)\}$. But every path here contains syntax errors and the result forest would be empty instead of containing two trees. We should choose another set (for example $\{(L_1; M_2); (L_2; M_1)\}$) to get the correct result.

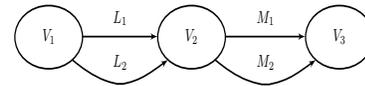


Fig. 8. Graph for minimal paths set selection.

So, path calculation is an iterative process. We perform state filtering during syntax analysis for each vertex with multiple input edges. Let describe the steps of the process:

- **Initial state.** Set of states for the vertex is empty.
- **Step.** For each step if the current vertex has multiple input edges then we should add new state to a state set for the current vertex if one of the following conditions is true:
 - new state corresponds to a path, which contains some edges which are not contained in any of the paths, which correspond to any state of the currently processing set;
 - new state corresponds to a parser state which is not yet presented in the currently processing set.

A pseudo code for the described algorithm is presented below.

```

/*
V list of input graph vertices in the topological order.
v_s start vertex of input graph.
*/
let filterStates v =
  let groupedByParserState =
    v.States.GroupBy (fun state -> state.Item)

  v.States = Set.empty

  for group in groupedByParserState do
    /* Each state corresponds with path from v_s to v.
    Set of paths specify set of edges of graph E_s.
    We should construct minimal set of paths which
    contains all edges of E_s. The next greedy algorithm
    can be applied to solve this problem.
    1) Order paths by length ascent.
    2) While current path contains edges which are not
    in the result set add this path in the result set.*/
    let ordered =
      group.OrderBy (fun s -> -1 * s.Path.Length)
    for s in ordered do
      if (s.Path contains edges which are
        not contained in any path corresponded
        with states from v.States || not s in v.States)
      then v.States.Add s

  for v in V do
    v.States <- /*step of syntax analysis*/
    /*If input degree of the vertex v more
    then 1 then try to filter states.*/
    if v.InEdges.Count > 1 then filterStates v

```

This way we can get state set which contains all parser states from input set but is not greater than it. Corresponding paths contain all possible edges in processed subgraph. Described algorithm of filtration allows to increase the performance of parsing by decreasing the number of parsing trees.

IV. EVALUATION

We implemented our algorithm of abstract translation in a tool built on top of FsYacc². We completely reused LALR generator, but implemented custom interpreter with stack copying ability.

Our tool was evaluated on a migration of a real-world project from MS-SQL Server 2005 to Oracle 11gR2. The original system contained 850 stored procedures and more than 3000 dynamic queries. The total size of the system was 2,7 million lines of code. More than half of all queries were complex; the number of query-generating operators varied from 7 to 212. The average number of query-generating operators was 40. We used PC workstation with Intel Core i7 2.6 GHz and 16 GB of RAM.

The results of comparison of two abstract translation implementations are presented in the Table I.

The first implementation was directly based on abstract parsing algorithm. That version was not adapted to process complex queries and turned system into active swapping. The analysis did not finish in acceptable time. Timeout (64 seconds) was added to limit one query processing time. Experiments showed that increasing timeout did not increase the number of processed queries. The number of queries, whose analysis was terminated by a timeout is shown in the table under the category "Dynamic SQL-queries with exponential growth of parsing forest".

The second implementation utilized state merging. State merging reduced the number of queries with exponential growth of parsing forest from 253 to 42, i.e. approximately in six times.

In the table below we present statistics for dynamic SQL query processing by two algorithms: original algorithm with timeout and algorithm with states merging.

Partially processed queries are those with non-empty parsing forest but with parsing or lexing errors. This category is the most difficult to deal with because error may be a false positive. Such situation may occur if query which triggers error can not actually be generated at run time.

V. CONCLUSION AND FUTURE WORK

Semantics calculation for embedded languages is also the source of problems. The main problem is that we cannot guarantee semantics correctness during syntax analysis: we can get correct tree with incorrect semantic. Example of this situation is shown on Fig. 9. In presented graph we can choose 2 paths which contain all variables used for query value calculation. For example, let we choose the paths which produce the next queries: "Select fld1 from myTbl1" and "Select fld2 from myTbl2". Both chosen paths are syntactical correct but in the real system the table myTbl1 may not contain the field fld1, and the table myTbl2 may not contain the field fld2.

Also we have problems which correspond with syntax of analyzes language and its specification in documentation and grammar. For example, such clauses of Select statement

TABLE I. COMPARISON OF THE ORIGINAL ALGORITHM WITH TIMEOUT AND THE ALGORITHM WITH STATE MERGING

Category description	Original algorithm with timeout	The algorithm with state merging
The total number of dynamic SQL queries	3122	3122
The number of successfully processed dynamic SQL queries	2181	2253
The number of partially processed dynamic SQL queries	408	522
Lexer errors	283	289
Parser errors	354	468
The number of not processed dynamic SQL queries	533	347
Lexer errors	140	134
Parser errors	280	305
Dynamic SQL queries with exponential growth of parsing forest.	253	42
Percentage of successfully processed dynamic SQL queries	69.86%	72.17%
Percentage of partially processed dynamic SQL queries	13.07%	16.72%
Percentage of dynamic SQL queries with non-empty forest	82.93%	88.89%

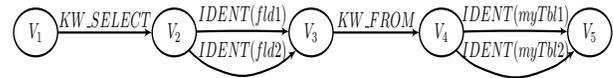


Fig. 9. All path in this graph are syntactical correct but semantics of some path may be incorrect.

as group by or order by. Any of these clauses can be omitted, but when the optional clauses are used, they must appear in the appropriate order and only one time per statement. But some simple approximation which allows to omit explicit enumeration of all variants of permutation is often used in the documentation and the grammar. Such approximation allows to accept input strings with arbitrary repetition of clauses (multiple repetition of one clause also possible). In the stored code such situation is not possible because this code should be correct but during graphs processing we can get Select query with multiple group by clause. This situation is not correct. The preferred solution of such problems is to use a special constructions in translation specification language. Also we can manually check correctness of parsing forest but this solution looks more difficult and less preferred.

REFERENCES

- [1] Shapot M., Popov E. Database reengineering // Open Systems.DBMS. Number 4. 2004.
- [2] Annamaa A., Breslav A., Kabanov J. e.a. An interactive tool for analyzing embedded SQL queries. Programming Languages and Systems. LNCS, vol. 6461. Springer: Berlin; Heidelberg. 2010. P. 131-138.
- [3] Annamaa A., Breslav A., Vene V. Using abstract lexical analysis and parsing to detect errors in string-embedded DSL statements // Proceedings of the 22nd Nordic Workshop on Programming Theory. Marina Walden and Luigia Petre, editors. 2010. P. 20-22.
- [4] Aske Simon Christensen, Miller A., Michael I. Schwartzbach. Precise analysis of string expressions // Proc. 10th International Static Analysis Symposium (SAS), Vol. 2694 of LNCS. Springer-Verlag: Berlin; Heidelberg, June, 2003. P. 1-18.
- [5] Grune D., Ceriel J. H. Jacobs. Parsing techniques: a practical guide. Ellis Horwood, Upper Saddle River, NJ, USA, 1990. P. 322.

²<http://fsharpowerpack.codeplex.com/wikipage?title=FsYacc>

- [6] Costantini G., Ferrara P., Cortesi F. Static analysis of string values // Proceedings of the 13th international conference on Formal methods and software engineering, ICFEM11. Springer-Verlag: Berlin; Heidelberg, 2011. P. 505-521.
- [7] ISO. ISO/IEC 9075:1992: Title: Information technology Database languages SQL. 1992. P. 668.
- [8] Java String Analyzer. URL: <http://www.brics.dk/JSA/>
- [9] Kyung-Goo Doh, Hyunha Kim, David A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology // Proceedings of the 16th International Symposium on Static Analysis, SAS09. Springer-Verlag: Berlin; Heidelberg, 2009. P. 256-272.
- [10] PHP String Analyzer. URL: <http://www.score.is.tsukuba.ac.jp/~minamide/phpsa/>
- [11] PL/SQL Developer. URL: <http://www.allroundautomations.com/plsqldev.html>
- [12] SQL Ways. URL: <http://www.ispirer.com/products>
- [13] SwissSQL. URL: <http://www.swissql.com/>
- [14] Xiang Fu, Xin Lu, Peltserverger B. e.a. A static analysis framework for detecting SQL injection vulnerabilities // Proceedings of the 31st Annual International Computer Software and Applications Conference. Vol. 01, COMPSAC07, Washington, DC, USA, IEEE Computer Society, 2007. P. 87-96.

Comparison of generalized ascent and descent parsers

Ragozina Anastasiya

Saint-Petersburg State University

Email: ragozina.anastasiya@gmail.com

Semyon Grigoriev

Saint-Petersburg State University

Email: rsdpisuy@gmail.com

Abstract—Syntax analyzers are used in many software reengineering tasks. This leads to create parser automatically. There are two basic types of parsers: ascent and descent. Descent parsers are popular because their control flow follows the structure of the grammar. However, descent parsers allow to process a very limited class of grammars. On the other hand ascent parsers allow to process a wider class of grammars (in particular left recursive grammars) but they are not so easy to understand and debug. Both classes suffer from the need to force the grammars to be in form which are deterministic, or at least near-deterministic for the chosen parsing technique. Nevertheless, generalised algorithms allow to remove these restrictions. The main problem posed in this paper is implementation of generalized table-based top-down parser algorithm and its comparison with generalized table-based bottom-up parser in the field of performance and errors detection.

I. INTRODUCTION

One of the main problem arising in the process of automatic software reengineering is a development of parsers [1] for programming languages. Syntax analysis may be used for translation, code analysis and other reengineering tasks.

Parsers may be separated into two classes: bottom-up and top-down. Each of them has advantages and disadvantages which are discussed below. Descent parsers (or top-down) are attractive because their structure is fully consistent with the structure of the grammar. Unfortunately, in spite of their readability, top-down parsers allow to handle a very limited class of grammars. The fact that LL(k)-grammars must be unambiguous causes strict restrictions on the languages [2]. It is typical for the naive implementation and generalised algorithms [6], [9] allow to get around this limitation. Left-recursive grammar are not LL(k)-grammar for any k. Sometimes it is possible to convert not LL-grammar to an equivalent LL-grammar by eliminating left recursion and factorization. However, the existence of an equivalent LL(k)-grammar for not LL-grammar is undecidable problem [3]. Backtracking methods [4] may extend the class of languages which may be processed by these analyzers but even it does not help to handle left recursion. However, left recursion problem can be solved by bottom-up (ascent) parsers. Bottom-up LR-analyzers[2] allow to handle a wider class of grammars in particular left recursive grammars. On the other hand, ascent parser do not have the direct consistency between them and the grammar that descent parsers have. Furthermore, LR(0) parse tables may be exponential in the size of the grammar [5] while LL-tables is linear. Moreover, even LR parsers can not cope with hidden left recursion. Also the performance of ascent analyzers is often

lower than performance of parsers constructed using top-down algorithm.

Both classes suffer from the need to force the grammars to be in form which are deterministic, or at least near-deterministic for the chosen parsing technique. Nevertheless, generalised algorithms allow to remove these restrictions. New generalised algorithm of top-down parsing was described in order to extend the class of languages processed by descent analyzers. Generalised LL (GLL) [6] handles all (including left recursive) context free grammars; runs in worst case cubic time; runs in linear time on LL grammars [2]. It also allows grammar rule factorisation, with consequential speed up. Most importantly, the construction is so straightforward that implementation by hand is feasible [10]. Parsers built using this algorithm may deal with both conventional and hidden left recursion and significantly extend the class of languages which are able to be processed by descent parsers.

II. OVERVIEW

As previously mentioned, syntax analysis plays an important role in software reengineering. Modern tools allow to generate parsers using specifications. Consequently, it is necessary to develop similar tools for the needs of automatic reengineering. The projects aimed to the development of tools for software reengineering automatization put forward specific requirements for parser generators [?] which are discussed further. Often the system to be reengineered are written on programming languages that have existed since the dawn of the development of the theory of syntax-directed translation. This means that the theory on which most of modern tools are based did not exist at that time. It leads to an impossibility of development of parsers for legacy languages using modern tools. In addition, parser generators for reengineering must handle a wide class of languages, allow to resolve ambiguities in the grammar, generate parsers with high performance and a good error recovery mechanism. Ease and convenience of the grammar specification language is another important feature which significantly influence on the development of parsers. It is essential for a specification language to make the parser creation process faster and make it easier to maintain parsers.

YaccConstructor [8], parser generator for the needs of automatic software reengineering developed at the Department of System Programming of the St. Petersburg State University, has all the mentioned features. YaccConstructor is a modularity tool which allows to create parsers using different specification languages and syntax analysis algorithms. Also, the tool supports its own specification language Yard. Yard allows to

use extended Backus-Naur form[2]. This form differs from the Backus-Naur form by its more "capacious" constructions which provide an opportunity to simplify grammar and reduce its size keeping the same expressive ability. Moreover, Yard language allows to create parameterized rules and to use a special syntax to resolve ambiguities.

A grammar is often changed in the process of reengineering. This changes make grammar ambiguous and lead to conflicts. This problem may be solved in different ways, for example, by using of GLR-algorithm. There is GLR-generator implemented as a part of the tool to deal with ambiguous grammars. It generates bottom-up parsers which use the RNGLR-algorithm [9]. Also error recovery mechanism and a mechanism which is provided an information about conflicts were implemented as a part of YaccConstructor project.

Previously, top-down parser could not provide opportunity to deal with ambiguous grammars but new generalized top-down parsing algorithm allows to handle ambiguous grammar and recursion. Generalized parsing algorithm is simple and it is also claimed that it has high performance. For these reasons, it was decided to implement new LL-generator using GLL-algorithm.

III. STATEMENT OF THE PROBLEM

The main problem posed in this paper is comparison of generalized bottom-up and top-down parsers. The following tasks were formulated to solve it:

- Implementation of a parser generator which uses GLL algorithm
- Implementation of a simple error detection mechanism
- Creation of the set of tests aimed to compare the performance of the generalized bottom-up and top-down analysis

IV. IMPLEMENTATION

There are two several approaches for automatic parser generation. It is possible to generate the whole parser code and then use it to build abstract syntax tree[2]. Schema of generator using this approach displayed in Figure 1.

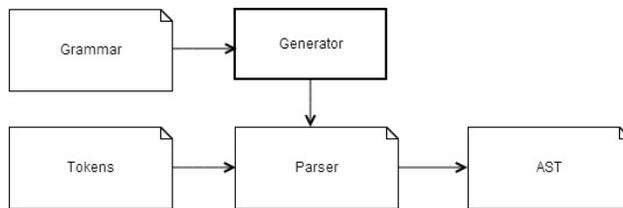


Fig.1. Scheme of parser generator creates whole parser code.

A different approach is a generation only additional information required for the interpreter. The interpreter is created in advance, contains main algorithm logic and it is reused without changes for every parser being generated. It uses additional information to build abstract syntax tree. This approach is more flexible because it provides an opportunity to creates several interpreters for different tasks. Scheme of such generator is displayed in Figure 2.

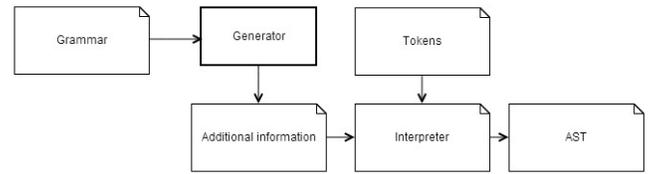


Fig.2. Scheme of parser generator creates only additional information.

V. DESCRIPTION OF THE ALGORITHM

The first approach is used in the article [6] which describes the algorithm which is taken as a basis. It consists of a set of functions for grammar nonterminals and control function. Control is passed between these functions by goto statements. In connection with the decision to use a different approach from the one described in[10], an algorithm has been adopted.

The input of the generator is grammar which is used to generate source file with additional information. This file is used in the future by the interpreter and contains the following additional information: grammar in the form convenient to process, functions for working with grammar and modified LL-table which is used to select rules for reduction. In the original algorithm, special functions are used to determine which alternative to choose instead of the table. Unlike a conventional table, LL-table may contain several rules in each cell. This situation arises because of the ambiguity of grammar.

Process of analysis has been also changed due to the rejection of the entire code parser generation. The parser from original articles consists of several functions corresponding to each nonterminal and one control function. Control function controls the process of analysis, checking whether the analysis was not completed and calling the necessary function. We use only two mutually-recursive functions (control function and processor) instead of several functions corresponding to each nonterminal. Control function performs the same role as before. There are some basic situations for the processing function:

- If x (currently considered symbol in grammar) is terminal then proceed to the next character in the rule and the input pointer is moved to next.
- If x is a nonterminal A then record current rule and position in it on the stack. This information is used to continue parsing after nonterminal x is processed to finish. A rule, by which x is revealed depending on the current character in the input stream, becomes a considered one. Pointer in the input stream remains unchanged.
- If a rule is considered to end and the current stack is not empty then pop the descriptor from the top of the stack and continue to work with these data.

Thus, the processing function simply performs a different action depending on the situation.

The generalized algorithm deals with ambiguity by mechanism similar to the mechanism of RNGLR. In ambiguous places it creates new processes each with its own stack. Special descriptors are used in GLL. Descriptors store label for goto

statement, stack and position in the input stream. Instead of labels we store just rule number and position in it.

Descriptors allow multiple configurations to represent processes in conflict situations. To do it in places where an ambiguity in the grammar is, new descriptors are created and pushed to the stack. The descriptors completely describe the current state of the process. The disadvantage of this approach is that for some grammars the number of descriptors depends exponentially on the size of the input. Another issue is that the mechanism cannot handle left-recursive grammars. To solve these issues the stacks are combined to graph structured stack (GSS) [6]. This structure allows to combine the stacks into a single one and record only one necessary vertex in descriptor. It significantly reduces the amount of memory needed for the algorithm.

It is necessary to make a number of control measurements in order to verify the effectiveness of the algorithm. After it, we should compare the results of existing generalized parsing algorithm and a new descent one on several criteria. It is assumed that the new algorithm shows better performance and more accurate data necessary to detect errors.

VI. RESULTS

Currently generator of additional data used for analysis is developed as a new module of YaccConstructor. Generated data contains a representation of the grammar, functions for working with it and modified LL-table, which mentioned earlier. There is GLR-module in YaccConstructor and many

useful structures are reused in GLL-module. For example, structures, allowing to store the grammar in a compact form, structures for abstract syntax tree building and etc. are reused. Recognizer based on GLL algorithm also is implemented.

REFERENCES

- [1] Alfred V. Aho and Ullman, *The Theory of Parsing, Translation and Compiling*, volume 1 - Parsing of Series in Automatic Computation. Prentice-Hall, 1972 , pages 33-45.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.
- [3] D.J. Rosenkrantz and R.E. Stearns. Proceeding STOC '69 Proceedings of the first annual ACM symposium on Theory of computing. ACM, 1969 , pages 165-180.
- [4] Dick Grune and J. H. Criel Jacobs. *Parsing Techniques: A Practical Guide (Second Edition)*. Springer, 2008.
- [5] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Criel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design (Second Edition)*. John Wiley & Sons, 2010.
- [6] Elizabeth Scott and Adrian Johnstone GLL Parsing. *Electronic Notes in Theoretical Computer Science* 253 (two thousand and ten) pages 177-189.
- [7] Y.A. Kirilenko, S.V. Grigoriev, D.A. Avdyukhin. Syntax analyzers development in automated reengineering of informational system. *Scientific and technical statements SPbSPU Issue 3 (174) / 2013*.
- [8] YaccConstructor home page <https://code.google.com/p/recursive-ascent/wiki/YaccConstructor>
- [9] Elizabeth Scott and Adrian Johnstone Right Nulled GLR Parsers.
- [10] Elizabeth Scott and Adrian Johnstone Modelling GLL Parser Implementations. *Engineering Lecture Notes in Computer Science Volume 6563* , 2011 , page 42-61.

One Approach to Automated Compiler Verification

Vyacheslav A. Bessonov
Department of Software and Computing
Systems Mathematical Support
Perm State University
Perm, Russian Federation
E-mail: v.bessonov@hotmail.com

Scientific Advisor:
Lyudmila N. Lyadova
Department of Business Informatics
National Research University Higher School
of Economics
Perm, Russian Federation
E-mail: LNlyadova@gmail.com

Abstract. Most modern software is written in high level languages. The task of translating source code, written in high-level languages, into a representation, which can be executed on a computer system, solves by specialized programs called compilers. Errors in compilers lead to differences between the behavior of modules, resulting from the work of compilers, and behavior, defining the semantics of the original program. Such errors are very difficult to detect and correct, and their presence casts doubt on the quality of the programs generated by a compiler. Obviously, the correctness of the compiler is a strong prerequisite for reliable software created with its help [20]. This paper describes the concept of a system designed to automate the process of testing the major components of any compiler: syntax analyzer and context conditions analyzer (semantic analyzer).

Keywords – compiler verification, automated testing, syntax analyzers testing, semantic analyzers testing

I. INTRODUCTION

All kinds of methods of software verification can be divided into two large groups [8]:

1. Static verification methods, including formal methods, methods of static analysis and expertise. Using of such methods implies that the verification of software systems is done “statically”, i.e. without execution on a computer system.
2. Dynamic methods that are used to verify the behavior of the program during execution.

The compiler of any language, having practical value, is such a complex system that static verification techniques can be used only for its individual small subsystems. Despite the fact that there are exceptions such as CompCert or π VC, common practice for compiler testing is dynamic verification [20], which involves the following tasks [14]:

1. Test generation (test writing).
2. A verdict on the results of test execution which is performed by the so-called test oracle, which is a procedure for determining the correctness of the system under this test.

3. Assessment of the tests quality which is performed with special test coverage metrics.

Currently, there are two common approaches used to solve these problems:

1. “White box” testing that used to identify all erroneous fragments of specific implementation.
2. “Black box” testing, designed to determine formal specification’s degree of compliance.

Model-based testing is a compromise between these two methods. This approach combines the advantages and eliminates the disadvantages of the above methods [20]. The model can be described formally, that allows using it as input for test generation and evaluation of test coverage. At the same time, the model defines the requirements for implementation and therefore it can be used to test the correctness of a particular implementation.

But it is obvious that manual construction and maintenance of the test suite is extremely difficult task. To simplify this task, it is proposed to use one of the main advantages of the model-based testing – the ability to systematically and automatically generate test cases [2]. The existence of a formal description allows automating the process of tests construction, which significantly reduces labor costs, and the systematic nature of testing increases confidence in its results.

Thus, the described above problems of the dynamic compiler verification can be summarized to the following problems [20]:

1. Automation of test construction:
 - a. Automating the generation of test data.
 - b. Automating the validation of test data processing (the problem of constructing a test oracle).
2. Determining the termination criterion of verification process.

In [20] verification scheme was proposed that designed to solve these problems. Its schematic representation is shown in Fig. 1.

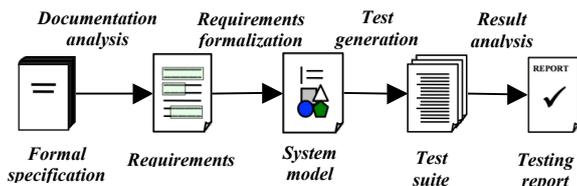


Fig. 1. Verification method scheme

The first stage of the scheme is requirements extraction from regulatory documents (e.g., specifications of the target programming language) and its classification. At the second stage a formal model is built via description of extracted requirements in some formal language. At the third stage test generation is performed on basis of the created model. It is often assumed that the user can optionally specify the desired size of the test suite, and/or test suite requirements in terms of some test coverage metrics. Depending on the task in addition to texts in the target programming language test suite may additionally contain an oracle for automatic verdict of the compiler correctness. At the last stage the created test suite is performed. After that reports on the entire process of testing are built. These reports contain information on how the compiler's observed behavior corresponds to the created formal model.

As mentioned above, compilers for real programming languages are extremely complex software systems. Furthermore, there is an additional source of difficulty in verifying compiler. It is the complexity of input data structure and its internal links. Obvious solution to reduce the complexity of the compilers verification task is functional decomposition into separate subtasks that should together cover all the functionality of the compiler [20]. Additional incentive for it is that the compiler is usually represented as a set of functional modules, which interactions and sequence are strictly defined.

However, in this paper it is considered verification of only the first three modules: lexical analyzer, syntax analyzer and semantic analyzer. It is worth noting that the development of the lexical analyzer often regarded not as a standalone module, but as an internal infrastructure for syntax analyzer. Under the semantic analyzer in the future will be understood analyzer of static semantics given by the set of so-called context conditions, as an example of which is the enforcement that all used variables are declared in the program code.

Thus, in accordance to the aforesaid task of compiler verification may be divided into the following subtasks:

1. Syntax analyzer verification.
2. Semantic analyzer verification.

In the case of automated testing, these tasks can be formulated as follows:

1. Syntax analyzer automated testing.
2. Semantic analyzer automated testing.

II. SYNTAX ANALYZERS AUTOMATED TESTING

The syntax analyzer is one of the core modules of any compiler and its incorrectness makes futile testing the rest of the modules. Therefore, verification of the syntax analyzer is one of the most important tasks of verifying compiler.

Positive tests generation

Since the 60's of the 20th century, many authors have investigated the grammar-based test generation for syntax analyzers.

One of the first works in the field was the work of Hanford [6], who proposed a method based on a "dynamic" grammar for generating test data for PL/1 compiler. Its drawbacks are the lack of any coverage metrics and non-deterministic nature of the method.

Purdom's work [15] considered fundamental. It contains one of the first coverage criteria for positive test sets: in a whole variety of tests for each grammar rule there must be language sentence, which is used in the derivation of this rule. In addition, in the same paper, the author proposed an algorithm for constructing a minimal test set that would satisfy this criterion.

Lämmel [10] showed that the Purdom's criterion is inadequate: tests that are constructed by this algorithm fail to detect the simplest errors. Stronger criterion proposed by Lämmel avoids this disadvantage and consisted in the fact that the test should cover each pair of rules, one of which can be applied directly after the other.

Many authors ([11], [12], [13]) proposed probabilistic methods of test generation. But in any case, this means that there is no guarantee that the algorithm has finished for the end time and thereby violates one of the basic principles that we have tried to follow, is consistency.

Negative tests generation

The above-described methods devoted exclusively to the generation of positive tests. At this time works, which would have offered methods for generating negative tests, are virtually absent.

A so-called "mutation testing" method is proposed in [7]. The basis of this method is the assumption that after the adding to the original grammar a number of changes (mutations) it can be used to generate potentially negative tests. However this approach entails the following problems:

1. Grammar-mutant can be equivalent to the original grammar.
2. Tests, generated on the basis of grammar-mutant, which is not equivalent to the source, may not be valid.

In [19] authors described methods for generating positive and negative tests and their coverage criteria. The authors embodied developed methods in the tool SynTesK. Using of this tool for testing industrial compilers confirmed the practical applicability of the developed approaches.

SynTesK main advantages are:

1. It is made under a unified methodology UniTesK, which formalizes the process of testing not only syntax analyzer, but also any other software.
2. Mechanisms of its work are based on the formal theory having a clear rationale.
3. It has open nature and is distributed with source code.
4. SynTesK allows to store together with tests their descriptive metadata (for example, the parse tree), which can be used for subsequent analysis.
5. Tool's functionality can be expanded through the development of specialized plugins.
6. The tool has real-world examples of successful application in practice.

SynTesK has the following disadvantages:

1. SynTesK allows using as a meta-language for formally describing the grammar only one certain type of EBNF. Users who use specialized tools to generate the syntax analyzer (Lex/Flex, SableCC, ANTLR, etc.) will be forced to perform translation from tool's meta-language to SynTesK meta-language.
2. It does not contain any specialized tools for managing sets of tests and analysis. SynTesK provides no opportunities to work with the generated tests (e.g., edit or delete), and the user is forced to use for this a file system, which greatly complicates the tests processing. In addition, it is often necessary to analyze a set of generated test (for example, to estimate the coverage metrics or determine the number of tests for a certain grammar rules, etc.), but SynTesK also provides no any special features for this and the user is forced to perform these operations manually.
3. SynTesK does not provide any special features to perform syntax analyzers profiling. For example, changing of string handling internal mechanisms in the syntax analyzer can strongly affect both the value of consumed memory and performance.
4. The tool interprets negative tests as a self-checking. However, apart from establishing the fact of error there must also ensure that the syntax analyzer correctly identifies the type of error and its location. Because application developers will use exactly this information when working with the compiler.

III. SEMANTIC ANALYZERS AUTOMATED TESTING

In their works Hanford [6] and Purdom [15] described the methods used to generate a positive tests for the syntax analyzers of procedural languages compilers, but these methods does not take into account any contextual conditions.

In [17] Wichmann and Jones proposed a method for constructing test sets, which would take into account some contextual conditions such as a correct processing of restrictions on the depth of nesting blocks, procedures blocks, cycles, etc. However, this method does not allow for other simple rules of static semantics, for example, concerning the using of variable names.

Celentano et al in [3] described the practical application of approach, which allows partially automate the testing of Pascal compiler. They used Purdom's algorithm to generate positive tests. To generate the test programs, which correct from the standpoint of static semantics, they used a specialized module with a grammar, augmented with a code for converting syntactically correct programs to semantically correct. The authors noted that the description of the context conditions in this way requires considerable effort and it is unlikely that this approach would be viable for testing modern programming languages analyzers.

In [5] authors offer to use attribute grammars as a formalism to describe contextual conditions. The resulting test suite, generated in accordance with the method proposed by Duncan and Hutchison, should contain only syntactically correct tests satisfying the context conditions. This is achieved by sequential scanning of all grammar production rules, which are executed only if it's permitted by contextual conditions. The tests generated by this method, should cover all grammar production rules and all described contextual conditions. However, this approach leads to a large number of empty runs of the generator, because of necessity to interrupt the process of generation due to unfulfilled contextual conditions. Furthermore, this approach leads to the construction of large numbers of semantically uninteresting tests [5].

In [16] Siner and Bershad described language Lava. Grammar defined on Lava reminds EBNF-grammar augmented by Java code describing the contextual conditions. The authors used Lava to generate a small number of tests (approximately 6 tests) with large size (approximately 60,000 instructions). These tests allowed making some resistance checks of Java Virtual Machine. Unfortunately, the paper does not give any estimates of test coverage.

In [1] author provides a method for constructive description of static semantics, as well as the method of generating both positive and negative tests. In addition, the author proposed a set of coverage criteria. The SemaTesK tool is the practical embodiment of proposed approaches.

SemaTesK as a SynTesK was developed in accordance with the methodology UniTesK and therefore inherits many advantages of this tool. Its other advantages are:

1. The tool uses an algorithm of semantically controlled generation. This algorithm makes it possible to systematically generate test data.
1. The performance of this tool is significantly higher compared to the other instruments (both real and hypothetical) [2]. It is achieved through the use of constructive test generation techniques.

Many SynTesK disadvantages, listed above, are also present in the SemaTesK. Its other disadvantages are:

1. One of necessary steps when working with the tool is the stage of creating a TreeDL representation of AST. However, in the case of using of specialized tools for the generation of syntax analyzers, this representation may be generated by this tool. For example, ANTLR generates a similar representation together with generation of grammar listener or visitor.
2. Users of the tool must create a specialized Java code intended for translation TreeDL representation into the text.

Common SynTesK and SemaTesK problem is that for the user they look like two completely different programs, each of which has its own characteristics and specific sequence of actions. For example, SynTesK user only has to run the program, passing to the input a formal description of the grammar and generation parameters. In the case of tool SemaTesK sequence of actions is much more difficult. In addition to formal description of context conditions user should also create TreeDL representation and develop Java code that performs mapping from TreeDL representation into the text. In the first case, a tester without any programming skills could handle the task of generating. In the second case, the requirements for the qualification of the tool's user are significantly higher.

IV. ANOTHER COMPILER TESTING SUITE

Our goal is to develop a system that would combine the advantages of the above-described tools and thus would be deprived of their disadvantages. First of all, the system must meet the following requirements:

1. Unified approach to test generation for syntax and semantic analyzers.
2. Presence of specialized tools designed to manage test sets and to analyze them.
3. Ability for integration with existing development tools using to automate the development process of syntax and semantic analyzers.

The system was called ACTS (Another Compiler Testing Suite) and its schematic representation is shown in Fig. 2.



Fig. 2. Automated testing system scheme

Components of this system are:

1. *Test generator* is the main component of the system. It is designed to automate the process of developing test sets.
2. *Test warehouse* is storage for test suites and their metadata. This component contains special tools for analyzing the repository content.
3. *Test runner* is a component, the main purpose of which is to automatically run test suites and collect the results of testing.

A. Test Generator

Test generator should use a unified approach to the generation of tests for both the syntactic and semantic analyzers. To implement this requirement, we suggest the following:

1. To use as a meta-language for grammar formal description a meta-language used in some of the most popular tools for generating syntax analyzers (for example, ANTLR).
2. To eliminate the need for intermediate TreeDL representation and use as a representation for the parse tree grammar classes generated by ANTLR tool. This, in turn, saves us from having to write additional code that performs the mapping from TreeDL representation into the text.

Schematic representation of the test generator is shown in Fig. 3.

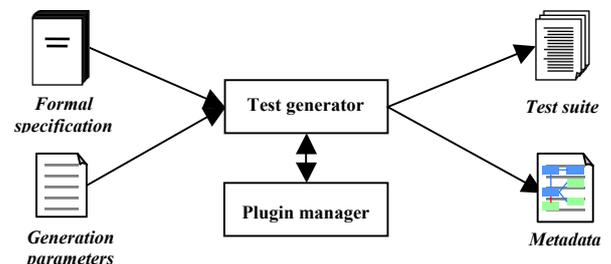


Fig. 3. Test generator scheme

Input data for the test generator are the formal specification of interesting language constructs and user-defined generation parameters. User may specify tests kind (syntax/semantic, positive/negative), test generation method kind, coverage metric, etc.

Currently there are many different methods for generating test data for syntax and semantic analyzers. Many of them are interesting from a practical point of view. That is why the test generator must provide the ability to use different methods of generation.

To implement this requirement, it is proposed to use the plugin-based architecture. Plugin is an abstraction of a method for generating tests and describes a generalized software interface that is used by the generator. Any

particular method of generation may be implemented as a separate plugin.

To control the individual plugins it is proposed to use specific module, called “plugin manager”. It allows viewing a list of available plugins, adding new or deleting an existing one. Test generator has access to a specific plugin only through the plugin manager. To select a specific plugin, the user must specify the appropriate information in the list of parameters passed to the input of the generator.

Schematic representation of the plugin manager is shown in Fig. 4.

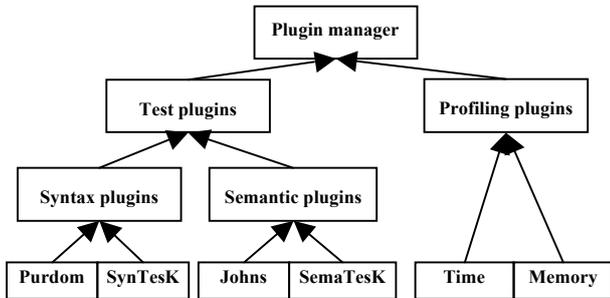


Fig. 4. Test generator plugin manager scheme

In addition to testing compliance of a developed analyzer to a formal specification, ACTS can be used to test analyzers efficiency and productivity. To do this, for example, ACTS can use specialized plugins designed for generation of tests with a very large number of instructions. These tests can be used for analyzers load testing. It is worth noting that these plugins do not have to be a stand-alone product and can use existing plugins for test generation.

The results of the test generator are test suite, which is a set of programs for a particular programming language, and set of metadata representing a formalized description of the test suite.

Such metadata can be extremely diverse. For example, such metadata can be a subset of the Dublin Core properties or the information of the tests structure.

B. Test Warehouse

Test suite and its metadata are placed in test warehouse. Testing reports are also stored in warehouse. Schematic representation of the repository is shown in Fig. 5.

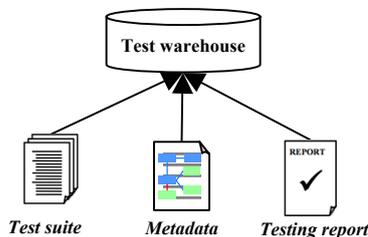


Fig. 5. Test warehouse scheme

In addition to direct physical storage warehouse should provide to the user with a convenient tools to control and analyze its content:

1. Test warehouse should provide a special opportunity to examine the contents of test suites and its metadata. For example, the user may need information on statistical information of existing tests: the number of positive/negative tests, the number of tests for a certain grammar rules, etc.
2. Warehouse must provide the ability to retrieve tests that meet certain criteria (for example, tests that verify the correctness of the implementation of a compiler module).
3. User should be able to view statistical information on the test results: the total number of uncorrected errors, common errors, etc.
4. Test warehouse may need also functions of version control system. At the case of new language development old tests can be an important historical material, showing the path of language development.

To implement this requirement, we propose to use warehouse’s structure, schematically depicted in Fig. 6.

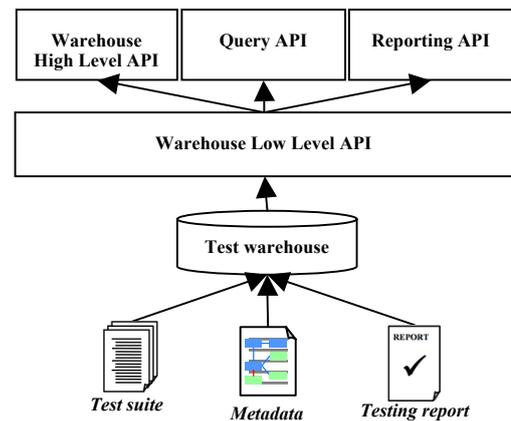


Fig. 6. Test warehouse extended scheme

Warehouse High Level API is a high-level programming interface for managing warehouse content (adding new, changing, or deleting an existing one) and for managing its different versions. The main purpose of this programming interface is abstracting from low-level operations like creating new repository, adding new file to repository, committing changes, etc., which would assumed working with specific version control system.

All low-level operations are performed by *Warehouse Low Level API*, which delegates the execution of these operations to a particular version control system. For example, Maven SCM API or specialized software interfaces used in different IDE (e.g., Net Beans VCS API).

Query API is a high-level programming interface for executing queries that retrieve various information from the

warehouse (for example, statistical information mentioned above).

Reporting API is a specialized programming interface for reporting. For example, this report is in addition to the standard information on the number of tests performed successfully or unsuccessfully, may also contain information extracted from the version control system (for example, information about what changes were made in the analyzer source code for a certain time period, by whom they were made and when).

C. Test Runner

Fig. 7 shows a schematic representation of the module running test suites. It is also based on the abstract program interface describing the runner, which can be used to run the tests in any programming language that are stored in the test warehouse.

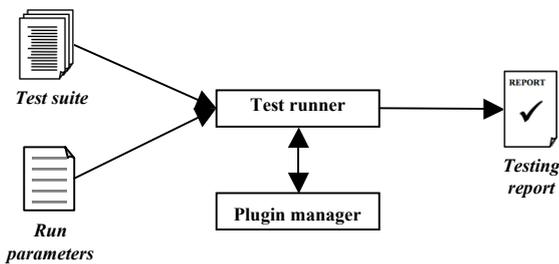


Fig. 7. Test runner scheme

Required possibility of extension, as in the case of the test generator, achieved through the use of plugins based architecture, where modules designed to run tests on a particular programming language acts as a plugins.

To work with plugins as well as in test generator test runner uses a specialized plugin manager, schematic representation of which is shown in Fig. 8.

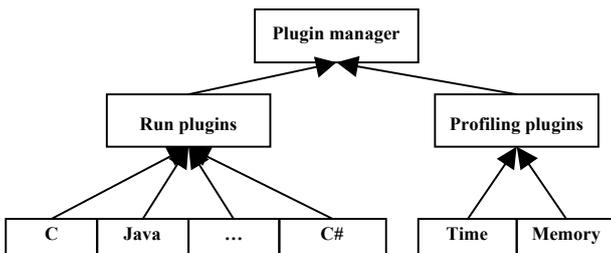


Fig. 8. Test plugin manager

In addition to plugins designed for running test suites and recording the results, ACTS must contain specialized plugins designed to perform profiling analyzers (for example, to determine the number of used RAM or to measure the total execution time).

The result of the test runner is the test report, which contains information on which of the tests have been passed,

and which are not, as well as any other information that may be needed for further analysis.

VIII. INTEGRATION WITH DEVELOPMENT INSTRUMENTS

As noted above, currently there are many tools designed to automate the development process of syntax and semantic analyzers: Lex\Flex, Yacc\Bison, SableCC, ANTLR, GOLD Parsing System, etc.

Pretty interesting scenario is the integration of tools that automates the creation of separate compiler modules and tools that automate the process of testing them. In this case, the resulting instrument would almost completely automate the entire process of developing a compiler or its individual modules and greatly facilitate the work of both developers and testers.

For example, in practice, it is not a rare case when one developed language is similar in many ways to others. “Language” at the same time may not necessarily mean a programming language (although in this case there are many examples of similarity of different languages, for example, C# and Java), but the description languages of different data structures, protocols, etc., or DSL languages. For example, the syntax grammar of the new DSL language may be based on the grammar of existing language, which has already been added to the warehouse. Thus the developer can create a new grammar, which includes existing rules and also the tests checking these rules. So with the help of a minimum set of actions developer can build not only a working analyzer, but also a set of tests that can be used to check how well the implementation meets the requirements.

For example, the ease of warehouse integration with different development environments provide a specialized abstraction level *Warehouse High Level API* which allows you to use any version control APIs that exists in modern IDEs (for example, Maven SCM API, NetBeans VCS API, etc.).

Using ANTLR in test generator should ensure ACTS easy integration in such a development environment like ANTLR Works or any ANTLR plugins, existing for other IDEs (IntelliJ IDEA, Eclipse and Visual Studio).

IX. CONCLUSION

In this paper it is introduced the concept of a system designed to automate the testing of syntax and semantic analyzers. The main advantage of this system compared to existing competing solutions:

1. Unified approach to test generation for syntax and semantic analyzers.
2. Presence of specialized tools designed to manage test sets and to analyze them.
3. Ability for integration with existing development tools using to automate the development process of syntax and semantic analyzers.

Together with instruments designed to automate the creation of separate compiler modules the system could almost completely automate the entire process of developing a compiler or its individual modules and greatly facilitate the work of both developers and testers.

A deep integration of testing tools and development tools can provide the high quality of the final product.

REFERENCES

- [1] Arkhipova M.V. "Avtomaticheskaya generatsiya testov dlya semanticheskikh analizatorov translyatorov", Dissertatsiya na soiskanie stepeni kandidata fiziko-matematicheskikh nauk. Moscow. 2006. ISP RAS.
- [2] Arkhipova M.V. "Generatsiya testov dlya semanticheskikh analizatorov", Vychislitel'nye metody i programmirovaniye, Vol. 7, 2006. pp. 55-70.
- [3] Celentano A., Reghezzi C.S., Della V.P., Granata G., and Savoretti F., "Compiler Testing using a Sentence Generator," Software - Practice and Experience, Vol. 10, No. 11, 1980. pp. 897-913.
- [4] CMMI for Systems Engineering/Software Engineering, Version 1.02 (CMMI-SE/SW, V1.02) CMU/SEI-2000-TR-018 ESC-TR-2000-018. 2000. pp. 598.
- [5] Duncan A.G., Hutchinson J.S. Using Attributed Grammars to Test Designs and Implementation // In Proceedings of the 5th international conference on Software engineering. Piscataway, NJ, USA. 1981. pp. 170-178.
- [6] Hanford K.V., "Automatic generation of test cases," IBM Systems Journal, Vol. 9, No. 4, 1970. pp. 242 - 257.
- [7] Harm J., Lammel R., "Two-dimensional Approximation Coverage," Informatica Journal, Vol. 2029, 2000. pp. 201-216.
- [8] Kulyamin V.V., "Integratsiya metodov verifikatsii programnykh sistem," Programmirovaniye, 2009.
- [9] Lämmel R., Verhoef C., "Cracking the 500-Language Problem," IEEE Software, Vol. 18, No. 6, 2001. pp. 78-88.
- [10] Lämmel R. Grammar Testing // Fundamental Approaches to Software Engineering. 2001. pp. 201-216.
- [11] Maurer P.M., "Generating test data with enhanced context-free grammars," IEEE Software, Vol. 7, No. 4, 1990. pp. 50 - 55.
- [12] Maurer P.M., "The design and implementation of a grammar-based data generator," Software: Practice and Experience, Vol. 22, No. 3, 1992. pp. 223-244.
- [13] McKeeman W., "Differential testing for software," Digital Technical Journal, Vol. 10, No. 1, 1998. pp. 101-107.
- [14] Posypkin M.A. "Primenenie formal'nykh metodov dlya testirovaniya kompilyatorov", Trudy Instituta sistemnogo programmirovaniya RAN, 2004.
- [15] Purdom P., "A sentence generator for testing parsers," BIT Numerical Mathematics, 1972. pp. 366-375.
- [16] Surer E., Bershad B.N. Using production grammars in software testing // In Proceedings 2nd conference on Domain-specific languages. New York, NY, USA. 1999. pp. 1-13.
- [17] Wichmann B.A., Jones B., "Testing ALGOL 60 compilers," Software - Practice and experience, Vol. 6, No. 2, 1976. pp. 261-270.
- [18] Yang X., Chen Y., Eide E., and Regehr J. Finding and understanding bugs in C compilers // Proceeding PLDI '11 Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011. pp. 283-294.
- [19] Zelenov S.V., Zelenova S.A. "Avtomaticheskaya generatsiya pozitivnykh i negativnykh testov dlya testirovaniya fazy sintaksicheskogo analiza", Trudy Instituta sistemnogo programmirovaniya RAN, 2004, Vol. 8.
- [20] Zelenov S.V., Pakulin N.V. "Verifikatsiya kompilyatorov – sistematicheskij podkhod", Trudy Instituta sistemnogo programmirovaniya RAN, 2007.

Generation of overlapped executable code

V. Aranov

Institute of applied mathematics and mechanics
SPbSPU
Saint-Petersburg, Russia
vladik@d-inter.ru

A. Terentiev

Institute of applied mathematics and mechanics
SPbSPU
Saint-Petersburg, Russia
alterterrific@gmail.com

Abstract—The paper discusses opportunities of new methods for creation modification resistant obfuscated executable code. In particular, overlapped executable code generation for CISC processors without instruction alignment with active use of custom modified LLVM framework is analyzed. The feasible approach for modification resistant code generation is proposed and a criterion for the quality estimation for proposed code transformation is provided. How this code generation approach can be used only for platforms not requiring strict instruction alignment like Intel x86 and x64 and cannot be employed for architectures like ARM, which have every instruction of the exactly same size and every instruction must be 4 byte aligned.

Keywords—*obfuscation; LLVM; code transformation; code generation; reverse engineering*

I. INTRODUCTION

The two main approaches to overcome software piracy threats are used: administrative one, including legislation support and organization piracy countermeasures and technical, which include different kinds of DRM, registration keys, software activation technologies and so on. We are to concentrate our efforts on the technical aspect of this problem.

The need of new code transformation in the industry is clear: a lot of pirated software available in the Internet shows inefficiency of current technical protection methods and techniques. There is no need to go far away to find examples. The first KMS activator for Windows 7 appeared in less than 3 months after operating system gone alive. For Windows 8.1 the same took place less than one month: in Oct. 17, 2013, the OS was published and around Oct. 25 KMS [1] activation solution was readily available for everybody to download in the Internet [2]. The client activation code for MS Windows starting with Windows XP uses an asymmetric cryptography, so it is impossible to generate the valid activation response. However, the valid KMS server can be bought by a client for local activation and the code from it was used to create KMS activator back in 2010 and 2013 years. No need to tell KMS client and server codes in both products were protected with anti-debugging techniques and properly obfuscated, but reality tells us “not enough did”. This is the most notorious case which calls us for new code generation methods for code execution in insecure environment.

Another example is WinRAR – a popular data compression product. Key-code generation algorithm or specifically private key for registration verification code was never publicly available, but counterfeit copies of WinRAR are still available

despite of all measures taken by Eugene Roshal and his team. The reason is simple: the code is either patched to ignore key code check at all (loosing archive authentication feature), or public part of registration checking part of the executable code is replaced with one in keygen [3].

These examples demonstrate the need for patch-proof code that cannot be easily modified by either third party or legal customer of the product. Current obfuscation technologies include mostly virtual machines, different morphing technologies, garbage code insertion and code encryption with runtime decryption coupled with heavy anti-debugging technologies [4], but every encrypted code has to be decrypted before execution and therefore can be modified. In addition, most anti-debugging technologies are well known; morphing and garbage insertion do not prevent code modification at all. Obfuscation virtual machines still provide serious challenges for hackers, but still could be defeated with enough efforts. So, something completely new should be invited. Overlapped code is promised to be one of such solutions, since it can be flawlessly integrated into general solution proposed in work [4].

II. OVERLAPPED CODE

A. Attacker's model

From now on we are going to use Bruce Schneier archetypes [5]. Let's assume Eve as a person with malicious intention to modify a program developed by Alice. Alice has transferred to Eve full program consisting of executable modules, dynamic linking libraries and data files. Eve has full control over execution environment which means that she can:

- Modify any and every byte of executable program at any given time.
- Set breakpoint at the any point of Alice application.
- Perform full snapshot of all address space Alice application is running in.
- Record execution traces.
- Perform backtrack debugging.
- Alice cannot react to Eve actions.

Therefore, Eve is like omnipotent Supreme Being relative to Alice code. However, no Eve actions except for the first one breaks execution logic of Alice code. While modifying the code, Eve supposes she does not break the logic of other parts

of the code except for that were just modified. However, two technologies break this assumption: making check sums and overlapped code.

Unfortunately, the code check sums are to defeat: many platforms have hardware “Page guard” [6] breakpoints to assist Eve. “Page guard” breakpoint is triggered only when CPU reads specific memory page, but not when executes. Therefore, overlapped code is the only valid option.

B. Overlapped code idea

How one can make a patch-proof code in this case? At first, such task seems to be impossible as soon as Eve has full control over execution environment with specified capabilities. However, there is a way showed on Fig. 1.

		add al, 0a3h		call dword ptr[eax]			
89	50	04	a3	ff	d0	05	08
mov edx, eax				mov eax, 0805d0ffh			

Fig. 1. Overlapped code with 4 bytes overlapped and 2 bytes shift

Bytes on the Fig. 1 encode two sets of instructions at once for x86 architecture:

```

mov edx, eax
mov ax, 0805d0ffh
and
add al, 0a3h
call dword ptr[eax]

```

Patching any overlapped byte will implicitly change meaning of another instruction in other code execution path. If this code path is not discovered by Eve, such code change may even go unnoticed because the task of discovering all executing control paths is not solvable for arbitrary case. In most cases using common tools like IDA, Hex-Rays and OllyDbg second layer code will not be even discovered using static code disassembly analysis, which means this approach not only having unclear way to defeat but also being hard to detect.

III. OVERLAPPING CODE QUALITY

Before starting overlapping code generation it is important to define exact goals of such generation, i.e. define a criterion answering the question: which of two pieces of overlapped code of the same functionality is better.

Let's define requirements for such criterion with the following assumptions: P – is a program of n size generated by reference LLVM compiler, Q – is a program of m size generated by overlapped code generator with same functionality as P , x_1, \dots, x_m – each byte usage count in program code, $W_P(Q)$ – target quality measure:

- $\forall P, i = 1..n: x_i \equiv 1$. We assume reference compiler does not generate overlapped code.

- $\forall Q, m = n, x_i = 1, i = 1..n: W_P(Q) \equiv 1$.
- $\forall Q, m < n, x_i = 1, i = 1..m: W_P(Q) > 1$. We do not want a huge program size. The shorter program code, the larger $W_P(Q)$.
- $\forall Q, m > n, x_i = 1, i = 1..m: W_P(Q) < 1$. The larger program code, the lower $W_P(Q)$.
- The more overlapping bytes in the code, the larger $W_P(Q)$.

In such case the suitable criterion will be:

$$W_P(Q) = \sqrt[d]{\frac{n \cdot \sum_{i=1}^m (x_i^d)}{m^2}}, \quad (1)$$

where d – is an arbitrary float parameter from $d \in (0, +\infty)$, where $d = 0$ means we do not care about overlapping at all and $d \gg 0$ means we prefer overlapping over the code size.

In general, the more $W_P(Q)$ value, the better result.

IV. GENERATION OF OVERLAPPED CODE

The ROP (Return-oriented programming) [7] technique had been employed for overlapping code generation task. This technique uses control over an exploited program to execute an arbitrary code in vulnerable application. However, we are to employ this technique for good. ROP defines sequences of instructions ending with `ret` or `jmp` instruction called *gadgets*. It is worth to mention, any instruction capable to modify instruction pointer register can be used as gadget finish instruction. According to ROP, the gadgets are usually searched in an application executable code or in dynamically linked libraries.

During ROP attack, Mallory [5] usually overwrites executing program stack and creates gadgets library. The first is not important for us and explored by R. Hund [8], but the latter is the way to go for our purpose. Let's consider two major ways to create a gadget library:

- Explicit instruction sequences. Explicit sequences are widely discovered in standard library functions. According to Roemer [7], libc library contains more than 4000 different potential gadgets capable to implement almost arbitrary algorithm, while the library size is only 1.3 Mbytes. However, explicit sequences are not important for us because of not increasing criterion (1).
- Implicit instruction sequences. These are instruction sequences we are looking for, since each byte these instructions consist of will increase (1). There sequences are obtained through looking for specific byte (or bytes) in the code (for example: 0c3h – `ret` instruction) and backward disassembly starting with this specific byte. One such byte(s) can usually produce more than one gadget. This approach would provide even more gadgets than explicit instruction case. However, one should be accurate with relocation item addresses. Fig. 2 provides good example of implicit gadget.

f7 c7 07 00 00 00	test EDI, 7h
0f 95 45 c3	setnz EBP
<hr/>	
c7 07 00 00 00 0f	mov EDI, 0F000000h
95	xchg EBP, EAX
45	inc EBP
c3	ret

Fig. 2 Implicit gadget example

The main difference from standard ROP is that initially we do not have any code to create gadgets from, because our compilation unit is empty. The *MakeOverlappedCode* algorithm is proposed to get around this problem:

In: $funcs = array[n]$ of *ByteFunction*
 Out: $newFuncs = array[n]$ of *ByteFunction*
 Algorithm:
 $gadgetList = nil$
 $newFuncs[0] = funcs[0]$
 for $i=1$ to n do
 $FindNewGadgets(gadgetList, newFuncs[i - 1])$
 $newFuncs[i] = InsertGadgets(funcs[i], gadgetList)$
 end for
 ,where *FindNewGadgets* has following pseudo code:

In/Out:
 $gadgetList = array$ of *Gadgets*
 In:
 $f = ByteFunction$
 Algorithm:
 for $i = 0$ to $sizeof(f)$ do
 if $f[i] == ret$ then
 //Add gadgets ending with i^{th} byte
 $FindGadgets(i, maxGadgetLength, gadgetList)$
 end if
 end for

Function *FindNewGadgets* looks for all bytes with specific instruction codes (*ret* in this example) in machine bytes forming function *f*. If specific byte sequence has been found, all byte sequences ending by this instruction are disassembled (backward disassembly). Disassembly is considered being successful if the last byte of disassembled instruction sequence is byte $[i]$. If disassembly was successful, the disassembled instruction sequence is added as a gadget into *gadgetList*.

MakeOverlappedCode works on function-based level with the following next steps:

a) For very first function in compilation module the code generated as usual with a normal LLVM codegenerator, however no new .CODE section is created for each function to disable function-level linkage and to enable cross-function gadgets. For the same purpose alignment bytes are not inserted between functions.

b) Inside every generated function new gadgets are discovered and added to *gadgetList*.

c) For every gadget added this way its LLVM representation pattern is created and added to instruction list to enable this gadget to be used as a normal instruction in every suitable case.

d) Finally instruction selector priorities are being manipulated to force instruction selector choose gadget type instructions over ordinary ones.

The greedy approach is used while inserting gadgets into newly generated code: if we can insert longer gadgets, we continue adding first suitable instruction into gadgets as much as possible. Such approach could potentially lead to miss of longer gadgets. However, experiments do not show big loss of the criterion (1) value, while avoiding exhaustive search is very important. As soon as we can't add more instructions to match our gadget, we completely remove generated gadget code, replacing it with call or jump to the found gadget.

It is possible to estimate time complexity of the approach. Disassembly of a limited sequence takes $O(1)$, the gadget list creation – $O(n^2)$. Insertion gadgets into the code – $O(n^3)$. Therefore, in the worst case the total time complexity of all actions performed is $O(n^3)$.

V. PRACTICAL RESULTS EVALUATION

The proposed approach has been evaluated using LLVM stress test kit. More than 1000 different programs has been generated and compiled using the standard LLVM code generator and our code generator enhanced with the approach proposed in this paper. The results are shown on Fig. 3 and 4. Value of criterion (1) here is the average value for all sample programs compiled.

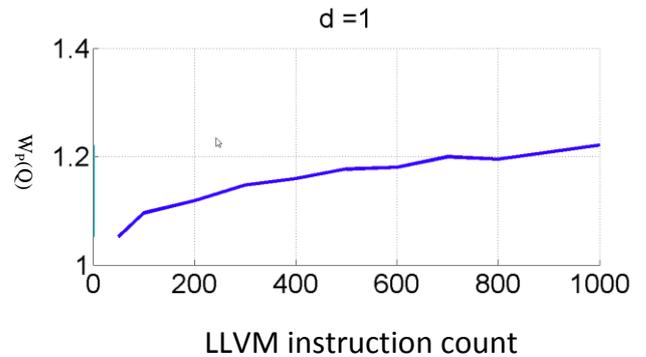


Fig. 3 Dependence of $W_p(Q)$ from compilation module size ($d = 1$)

For $d = 1$ (Fig. 3) we virtually prefer neither size of the program nor amount of instruction bytes being overlapped. Fig. 3 demonstrates that with such choice of criterion the quality of our code generator gradually increases with increase of the amount of the code being compiled. This is an expected result, because LLVM code generator has more probability to discover gadget in the code already compiled and more versatility of such gadgets discovered. However, the aggressiveness of gadgets usage is limited by the size of output data. To tune the algorithm it is possible to use other values of parameter *d*.

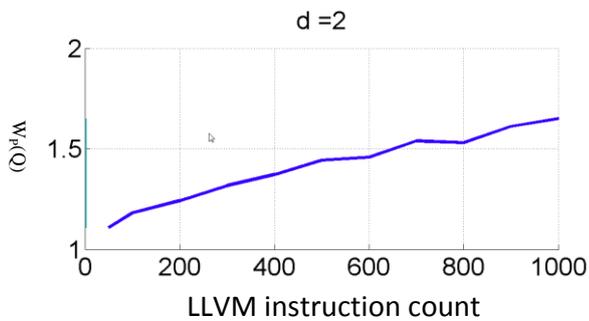


Fig. 4 Dependence of $W_p(Q)$ from compilation module size ($d = 2$)

For $d = 2$ (Fig. 4) we favor overlapped bytes more than size and our criterion benefits more from overlapped bytes rather than from decrease of program size.

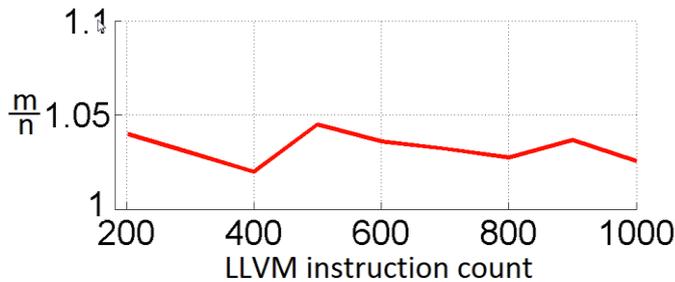


Fig. 5 Compiled program size reduction

It is noteworthy to tell that in some cases the proposed approach was able to produce code (Q) better than normal code produced by compiler (P) not only in terms of criterion (1), but in terms of the size in bytes too. This result was not intentionally pursued and appeared as the following positive side effect demonstrated on Fig. 5: the size of the program is 4% reduced in average after compilation. While the whole reduction is not big and depends on the actual code, it is still worth to save about 700 bytes for 19Kbytes (roughly corresponds to 1000 LLVM instruction program) of the compiled code.

TABLE I. PERFORMANCE OF THE OVERLAPPED CODE

Algorithm	Reference time, sec	Overlapped time, sec	Overlapped CPU cycles/Reference CPU cycles
Taylor series (sin)	3.628	3.574	1.0007
Factorial	3.334	3.477	1.0430
Fibonacci	3.301	3.211	1.0260

Measurements in Table 1 made for 1000 repetitions of the algorithm in the first column demonstrates insignificant performance slowdown in both CPU cycles and real execution times, while overlapped code sometimes executes even faster compared to reference code, probably due to the smaller code size in the overlapped code case.

VI. FUTURE WORK

The approach proposed by Joshua Mason [9] looks like the most prominent way to improve criterion (1) and makes better overlapped code. Since we are interested in increase of

criterion (1) we can use Viterbi algorithm [10] to traverse our collection of gadgets in conjunction with hidden Markov model to reconstruct most probable sequence of states used in HMM. Where each function being encoded in Markov model, which states specified by unknown parameters (most suitable gadgets or ordinary glue instructions in our case) and known parameters (list of gadgets we are already have).

Such approach would allow us to avoid greedy approach and has prominent potential to increase quality of overlapped code.

Usage of the proposed approach for compilation of size critical code for SOCs and microcontrollers is one of further research goals and can be further improved.

REFERENCES

- [1] Microsoft Windows volume activation reference guide, Microsoft Corporation, October 2009
- [2] Vlad Dudau, Windows 8.1 activation has been bypassed, 2013, URL: <http://www.neowin.net/news/windows-81-can-now-be-activated-with-kms-workaround-tool> (accessed on April 11, 2014)
- [3] Practical Reverse Engineering Tutorial - Cracking Winrar, 2011, URL: <http://www.hackingalert.net/2011/09/practical-reverse-engineering-tutorial.html> (accessed on April 12, 2014)
- [4] Aranov V.Y., Zaborovskiy V.S., Method of executable code protection from reverse engineering, Problems of information security. Computer systems, SpbSPU, p. 93-97, 2013.
- [5] Bruce Schneier, Applied cryptography (2nd ed.): protocols, algorithms, and source code in C, John Wiley & Sons, Inc., New York, NY, 1995
- [6] Gene Novark , Emery D. Berger, DieHarder: securing the heap, Proceedings of the 17th ACM conference on Computer and communications security, October 04-08, 2010, Chicago, Illinois, USA
- [7] Ryan Roemer , Erik Buchanan , Hovav Shacham , Stefan Savage, Return-Oriented Programming: Systems, Languages, and Applications, ACM Transactions on Information and System Security (TISSEC), v.15 n.1, p.1-34, March 2012 .
- [8] Ralf Hund , Thorsten Holz , Felix C. Freiling, Return-oriented rootkits: bypassing kernel code integrity protection mechanisms, Proceedings of the 18th conference on USENIX security symposium, p.383-398, August 10-14, 2009, Montreal, Canada.
- [9] Joshua Mason , Sam Small , Fabian Monrose , Greg MacManus, English shellcode, Proceedings of the 16th ACM conference on Computer and communications security, November 09-13, 2009, Chicago, Illinois, USA .
- [10] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. IEEE Transactions on Information Theory, 13(2):260--269, April 1967

Predicative analytics for developing software

Nadejda Yarushkina
Dept.of Information systems
Ulyanovsk state technical University
Ulyanovsk, Russia
jng@ulstu.ru

Tatiana Afanaieva
Dept.of Information systems
Ulyanovsk state technical University
Ulyanovsk, Russia
tv.afanasjeva@gmail.com

Irina Timina
Dept.of Information systems
Ulyanovsk state technical University
Ulyanovsk, Russia
i.timina@ulstu.ru

Abstract—The article is devoted to the problem of applying the formal data mining tool – forecasting – for the developing of new software and for reengineering the present software. We propose the algorithm adjustments of the time series forecasting. This algorithm takes into account the dependence of the current state of time series from the previous one, the influence of basic fuzzy projected trends in the time series. The proposed algorithm expands the opportunities of time series short-term forecasting on the base of fuzzy trends, as the historical software time series are of small length. The proposed algorithm was examined experimentally and showed the efficiency

Keywords—data mining, development, software, fuzzy time series, forecasting, fuzzy tendency

I. INTRODUCTION

Predicative analytics is the major formal tool for developing of new software and for reengineering the present software. To develop not only new, but competitive software, it is necessary to research of appearing trends by using formal methods at an analysis stage. This research should be directed on the perspective functions and software technologies, scientific achievements and user requirement. So, to fulfilled such research for development of the competitive software the data mining algorithms, extracting new trends and predicative analytics have to be used. The predicative analytics is the effective tool in the analysis of trends, if we want to create new competitive software. The core of the predicative analytics undoubtedly is the time series analysis of the important software parameters.

Forecasting is one of the problems of Time series analysis. The results of Time series forecasting and its trends are useful for business and management. Particularly the forecasting of

economic indicators and its trends is a part of planning process in the enterprise. Unfortunately, the trend forecasting, as a data mining formal tool, for developing of new software and for reengineering the present software practically isn't used. The regular analysis of the trend and dependencies in technologies, scientific achievements and user requirement, expressed in time series, allows to uncover ways to create a new useful ideas for software development.

There are two approaches to Time series forecasting and analysis. The first one is based on forecasting only one Time series. This approach is widely spread, and many methods and models have been proposed: statistical [1-4], fuzzy [5-9], and their combination [10-13]. In this approach regression models on time and autoregression models are used to predict values and global trends.

In The second approach to Time series forecasting the predictive model includes another Time series values in addition. In this approach it is supposed that variation of one time series causes the variation of another time series. To estimate this dependency the regression model has to be identified and to build the adequate regression model the cointegration between TS has to be studied.

If time series are not cointegrated, then the estimation of their dependency makes no sense. Fundamental results in this problem were generated by Granger, Engle, Johansen, Phillips in [14-16]. There are well-known methods for cointegration testing: Engle–Granger test, the Johansen test, Phillips–Ouliaris cointegration test. Cointegration is an important property of many economic indicators. To forecast them the ECM- time series modeling with correction errors was proposed [15]. The main idea of ECM is to correct model for short-term dynamics in accordance with long-term dependence between time series.

However, most of the methods are required long-term time series, and the time series might be the same length. The mentioned fact makes a serious problem for an applied researcher when forecasting time series of software indicators, particularly, in case time series are not long enough, for example, 20 values (short-term time series).

For the analysis of a dependencies between short-term time series fuzzy models could be used computational Intelligence techniques [7-9], [17], [19].

According to fuzzy modeling the components of time series model are considered as fuzzy sets, and there were proposed a lot of techniques. In [17] three groups of fuzzy time-series data models are considered: (1) a regression model-based analysis [6], [11] by using a fuzzy regression coefficient, (2) a Box-Jenkins model-based analysis by using a fuzzy autocorrelation coefficient [18] (3) a fuzzy reasoning (IF-THEN rule)-based analysis by using fuzzy time series model [7-8].

In [9],[12] the modification of fuzzy time series model is proposed for modeling fuzzy short-term tendencies.

However, the problem of modeling fuzzy tendencies (local trends) based on dependences between fuzzy time series, has received far less attention and is still an open problem. The importance of this problem lies in the fact that many time series of software indicators are interdependent and the situation that fuzzy trends of TS Z might be a predictor of TS Y is really exist.

In this paper we proposed a new time series forecasting algorithm, using fuzzy trends time series model and hypothesis about dependences between two time series. We embed explicit hypothesis from an expert of applied field, that fuzzy trends of time series Z is a predictor of fuzzy trends of time series Y. The term predictor is used for determination of significant convincing characteristics providing the most precise forecast of any phenomenon. In other words the predictor is a pre-requisite of a certain important event evaluated and being a part of a respective equation for the forecast.

The structure of the article is the following: Part 2 briefly describes basic provisions of the proposed time series forecasting algorithm; Part 3 shows theoretic basics of fuzzy trends time series model; Part 4 demonstrates the new algorithm of time series forecasting and computing experiment results.

II. BASIC PROVISIONS OF PROPOSED ALGORITHM

In the development of new software applications on the analysis of the IT market is one of the important tasks. Indicators of the IT market for some time period can be represented by time series. It should be noted the following features of the time series: short time series (less than 20 values), nonstationary time series, the existence of the relationship between time series, time series with missing values. The main purpose of market analysis software is forecasting the trends in sales volumes in different segments of the software applications.

Short length, non-stationary behavior, inaccurate data and problem of selection of a proper model are the factors that complicate the use of classic statistical models and methods [1-4].

The analysis and forecast of such time series is held usually by software experts, forming results in the form of linguistic terms of tendencies: Small growth, Rapid Fall, Stability.

It is known that to design linguistic values the fuzzy-set theory is used [5], which is the base for fuzzy TS forecasting models. Models for fuzzy values forecast are called fuzzy time series [8].

But the problem of fuzzy trends modeling is under-investigated.

A. Analysis of fuzzy trends definitions

In statistical approach the analysis of behavior characterizing long-term dynamics of TS is connected with the trend concept describing long-term dependence of time series values from time. So in accordance with [3] mathematic model of time series is represented in the form:

$$y_t = f(t) + u_t \quad (1)$$

Herewith it is assumed the presence of deterministic trend (or the aggregate of trends) upon the whole of time series.

Diversity of possible variants of TS trend component behavior approximate quantitative estimation and absence of opportunities to identify qualitative assessments caused the idea to use fuzzy trend concept.

In 1982 H. Tanaka [11] proposed the model of linear regression with fuzzy coefficient and applied methods of linear programming. However the use of fuzzy coefficients did not allow to solve the problem of qualitative TS trend identification.

Further research of FTS brought out a new task of FTS object description, modeling and forecasting – fuzzy trend as a representative of qualitative changes expressing the changes not in numerical but in fuzzy values of TS [7].

The sequence of FTS fuzzy trends in time dimension brings out fuzzy time series with fuzzy trend [9].

Our research showed that the elimination of the above mentioned restrictions by introducing the forecast correction procedure considering main trend allows to get a more accurate forecast of a future TS trend of a software market indicators.

III. THEORETICAL PROPOSITION OF PROPOSED TIME SERIES FORECASTING ALGORITHM

Suppose there are given a discrete time series $Y = \{t_i, x_i\}, i = 1, 2, \dots, n$. According to the basic provisions of FTS theory, developed by Zadeh [5], Song and Chissom [8], any finite discrete time series – numeric, nonnumeric, mixed – might be transformed into FTS $\tilde{Y} = \{t_i, \tilde{x}_i\}, i = 1, 2, \dots, n$, given that its value set $X = \{x_i\}$ will be covered by specific functions (fuzzy sets) $\tilde{x}_i \in \tilde{X}, j = 1, 2, \dots, m, m < n$.

Definition 1. Fuzzy trend τ , assigned on the segment $[t_i, t_j], t_j \geq t_i$ with \tilde{x}_i, \tilde{x}_j of fuzzy time series \tilde{Y} is a fuzzy term assigning fuzzy increment $\tau = \tau((t_i, \tilde{x}_i), (t_j, \tilde{x}_j))$.

Generalized model of FTS FT will be:

$$\tau_i = f(\tau_{i-1}, \dots, \tau_{i-d}) \quad (2)$$

where d – a fixed number, model parameter;
 $\tau_i, \tau_{i-1}, \dots, \tau_{i-d}$ – a sequence of fuzzy trends;
 f – some fuzzy dependence.

Substantial analysis allows to conclude that the term *Trend* defines qualitative changes upon the time domain and is used in sentences along with general linguistic assessments connected with content function, type and intensity, for example, Long Growth Trend, Strong Fall Trend, High Quality Stability Trend, etc.

Therefore it is worthwhile to mark out the following characteristics for FT:

- **Fuzziness.** Fuzziness is a fact that FT is built on the base of fuzzy values of FTS and inherits the fuzziness of these values, time series might correspond with various fuzzy trends with different grade of membership.
- **Duration.** Duration is a characteristic of various duration of fuzzy trend.
- **Typicality.** FT typicality property allows to discern classes, FT types, which have fuzzy trends considered as homogeneous within.
- **Significance.** For various fuzzy trends of one type and equal duration application of level of FT significance of intensity characteristics is appropriate.
- **Time awareness.** This property shows that fuzzy trends are determined between two values of time interval.
- **Linguistic interpretability.** This fuzzy trend property follows the definition of fuzzy trend as a quality changes characteristic. Fuzzy trend is defined as a fuzzy mark matching linguistic term.

We suggest a more detailed description of fuzzy trend which has fuzzy time series. For this purpose let the following statements and definitions be introduced.

Assume that the linguistic variables Fuzzy Time Series, Fuzzy Trend, Type_Trend, Intensity_Trend, Duration_Trend are assigned with basic finite term-sets $\tilde{X}, \tilde{\mathfrak{T}}, \tilde{V}, \tilde{A}, \Delta T$ respectively.

Definition 2. Each fuzzy trend $\tau \in \tilde{\mathfrak{T}}$ of fuzzy time series $\tilde{Y} = \tilde{x}_t, t = 1, 2, \dots$ might be shown a structure model in the form of relation tuple built on Cartesian product of fuzzy trend properties $\tilde{V} \times \tilde{A} \times \Delta T \rightarrow \tilde{\mathfrak{T}}$:

$$\tau = \langle \tilde{v}, \tilde{a}, \Delta t, \mu \rangle \quad (3)$$

where τ – a name of fuzzy trend from the set $\tilde{\mathfrak{T}}, \tau \in \tilde{\mathfrak{T}}$;

\tilde{v} – a type of fuzzy trend (change type) $\tilde{v} \in \tilde{V}$ shows basic quality dependences of time series {Fall, Growth, Stability}.

\tilde{a} – intensity of fuzzy trend, $\tilde{a} \in \tilde{A}$, might be introduced linguistically, e.g. values from the set {Intense, Average, Weak};

Δt – duration of fuzzy trend, $\Delta t \in \Delta T$;

μ – a membership function of a FTS segment bounded by interval Δt of fuzzy trend τ .

Classify fuzzy trends of fuzzy time series in accordance with duration into elementary $T \in \tilde{\mathfrak{T}}(\Delta t = 1)$ local $N_t \in N_{\tilde{\mathfrak{T}}}(1 < \Delta t < n - 1)$ and basic (general) $G_t \in G_{\tilde{\mathfrak{T}}}(\Delta t = n - 1)$.

Definition 3. Elementary fuzzy trend (EFT) of fuzzy time series $\tilde{Y} = \tilde{x}_t, \tilde{x} \in \tilde{X}, t = 1, 2, \dots, n$ is a fuzzy trend $\tau_t = \langle \tilde{v}_t, \tilde{a}_t, \mu_t \rangle$ showing the character of change of FTS segment between two neighboring fuzzy FTS marks $\tilde{x}_{t-1}, \tilde{x}_t$ with membership degree $\mu_t = \min(\tilde{x}_{t-1}(x_{t-1}), \tilde{x}_t(x_t))$.

Types of elementary trends are basic types of fuzzy trends of FTS from the set $\tilde{V}_1 = \{\tilde{v}_1, \tilde{v}_2, \tilde{v}_3\}$, \tilde{v}_1 =Stability, \tilde{v}_2 =Fall, \tilde{v}_3 =Growth.

Definition 4. Finite elementary trend $\tau_s = \langle \tilde{v}_s, \tilde{a}_s, \Delta t_s, \mu_s \rangle$ is an elementary trend built on the last pair of neighboring FTS values.

Definition 5. Elementary fuzzy trend (EFT) time series is introduced in the form

$$\begin{aligned} \tilde{v}_t &= TTend(\tilde{x}_t, \tilde{x}_{t+1}), \tilde{a}_t = RTend(\tilde{x}_t, \tilde{x}_{t+1}), \\ \mu_t &= \min(\mu(\tilde{x}_t), \mu(\tilde{x}_{t+1})) \end{aligned} \quad (4)$$

Statement 1. Any finite discrete time series might be transformed into time series of EFT.

Specify the generalized model of EFT time series and define main components EFT changes admitting that this model might behave differently.

Definition 6.

Let $X_t, (t = 1, 2, \dots) \subset R^1, R^1$ is a universe discourse, where fuzzy sets $\tilde{x}_t^i, (i = 1, 2, \dots), \tilde{v}_t^j, (j = 1, 2, \dots), \tilde{a}_t^s, (s = 1, 2, \dots)$ are defined and \tilde{X}_t is a collection of $\tilde{x}_t^i, (i = 1, 2, \dots), \tilde{V}_t$ is a collection of $\tilde{v}_t^j, (j = 1, 2, \dots), \tilde{A}_t$ is a collection of $\tilde{a}_t^s, (s = 1, 2, \dots)$. Let relations $R_V: \tilde{X} \times \tilde{X} \rightarrow \tilde{V}, R_A: \tilde{X} \times \tilde{X} \rightarrow \tilde{A}$ exist, then the model of fuzzy dynamic process with fuzzy differences is

$$\tilde{X}_t = (\tilde{X}_{t-1} \times \tilde{V}_t \times \tilde{A}_t) \circ R(t, t - 1) \quad (5)$$

where

$$\begin{aligned} \tilde{V}_t &= \tilde{V}_{t-1} \times \tilde{V}_{t-2} \times \dots \times \tilde{V}_{t-p} \circ R_{\tilde{v}}(t, t - p), \\ \tilde{A}_t &= \tilde{A}_{t-1} \times \tilde{A}_{t-2} \times \dots \times \tilde{A}_{t-q} \circ R_{\tilde{a}}(t, t - q) \end{aligned} \quad (6)$$

there $\tilde{X}_t, \tilde{X}_{t-1}$ is a state of fuzzy process, coded by fuzzy sets (linguistic terms);

$R(t, t - 1)$ – fuzzy relation defining the first-order model in terms of fuzzy values \tilde{X}_t , which can be represented by sets of fuzzy “equations” in the form of *IF-THEN*;

$\tilde{V}_t = \tilde{V}_{t-1} \times \tilde{V}_{t-2} \times \dots \times \tilde{V}_{t-p} \circ R_{\tilde{v}}(t, t - p)$ is p -th order fuzzy time series model of FT-type changes (changes type) shows basic quality dependences of time series {Fall, Growth, Stability},

$\tilde{A}_t = \tilde{A}_{t-1} \times \tilde{A}_{t-2} \times \dots \times \tilde{A}_{t-q} \circ R_{\tilde{a}}(t, t - q)$ is q -th order fuzzy time series model of FT intensity changes (changes intensity),

\circ – composition sing in fuzzy theory; $p > 0; q > 0$.

The model of the numeric time series $Y = \{t_i, x_i\}, (i = 1, 2, \dots, n)$ is represented in the form of:

$$x_t = x_{t-1} + v_t \cdot \alpha_t + \varepsilon_t \quad (7)$$

where x_t, x_{t-1} – numeric values of time series, generated by defuzzification of FTS fuzzy values $\tilde{Y} = \tilde{x}_t, \tilde{x} \in \tilde{X}, t = 1, 2, \dots, n$;

$x_t = deFuzzy(\tilde{x}_t)$, $x_{t-1} = deFuzzy(\tilde{x}_{t-1})$, $t = 1, 2, \dots, n$
 here v_t – numeric values, defining EFT type, obtained as a result of defuzzification of fuzzy values $v_t = deFuzzy(\tilde{v}_t)$;
 α_t – numeric value defining EFT intensity, obtained as a result of defuzzification $\alpha_t = deFuzzy(\tilde{\alpha}_t)$; ε_t – errors.

To defuzzificate fuzzy trend type the following formula is used

$$DeFuzzy(\tilde{v}_i) = \begin{cases} 0, & \text{if } \tilde{v}_i = \text{"Stability"} \\ -1, & \text{if } \tilde{v}_i = \text{"Fall"} \\ 1, & \text{if } \tilde{v}_i = \text{"Growth"} \end{cases} \quad (8)$$

To defuzzificate intensity the centroid method ($nmin$, $nmax$ – determined by min and max differences of TS values) is used:

$$DeFuzzy(\tilde{\alpha}_t) = \frac{\int_{nmin}^{nmax} x \cdot \tilde{\alpha}(x) dx}{\int_{nmin}^{nmax} \tilde{\alpha}(x) dx} \quad (9)$$

According to the approach suggested in this work the results of EFT forecasting should be corrected considering the main trend $G\tau$. To identify the main fuzzy trend $G\tau$ and to define its TS components we suggest heuristic algorithm where the applicable components are determined experimentally. To make algorithm function the source TS was transformed in FTS.

Algorithm 1.

Step 1. Deriving of fuzzy elementary trend time series $\tau_t = \langle \tilde{v}_t, \tilde{\alpha}_t, \mu_t \rangle$, $t = 2, 3, \dots, n$ and defuzzification of EFT intensities is according formula (9): $\alpha_t = deFuzzy(\tilde{\alpha}_t)$.

Step 2. Calculate the cumulative intensity of sane-type EFT upon the whole of time series $\tau_t = \langle \tilde{v}_t, \tilde{\alpha}_t, \mu_t \rangle$ ($t = 2, 3, \dots, n$):

$$\begin{aligned} & \text{IF } (v_t = \text{"Growth"}), \text{ THEN } ST_{growth} = ST_{growth} + \\ & \alpha_t, \mu_{growth} = \max(\mu_{growth}, \mu_t); \\ & \text{IF } (v_t = \text{"Fall"}), \text{ THEN } ST_{fall} = ST_{fall} + \alpha_t, \mu_{fall} = \\ & \max(\mu_{fall}, \mu_t). \end{aligned}$$

Step 3. If ($ST_{growth} = 0$ and $ST_{fall} = 0$) or ($ST_{growth} = ST_{fall}$), then the type of the main FT $\tilde{v}_{G\tau} = \text{"Stability"}$ and after defuzzification $v_{G\tau} = 0$, dynamics of time series is stationary, otherwise Step 4.

Step 4. On the base of comparative analysis of values ST_{growth} and ST_{fall} determine the type of main fuzzy trend. If $ST_{growth} \geq 2 \cdot ST_{fall}$, then $\tilde{v}_{G\tau} = \text{"Growth"}$ and after defuzzification $v_{G\tau} = 1$, otherwise the type of main fuzzy trend $\tilde{v}_{G\tau} = \text{"Fall"}$ and after defuzzification $v_{G\tau} = -1$. Time series dynamics is non-stationary.

Step 5. Then the main trend intensity is:

$$\alpha_{G\tau} = |ST_{growth} - ST_{fall}|.$$

For the model FTS (sample of 50 TS of short length), the accuracy of basic trend identification $G\tau$ suggested by the algorithm was 99 %.

IV. ALGORITHM OF SHORT-TERM TIME SERIES FUZZY TRENDS FORECASTING

Consider the algorithm of TS for casting $Y = \{t_i, x_i\}$, ($i = 1, 2, \dots, n$), on the assumption that expert's hypothesis that TS fuzzy trend $Z = \{t_i, z_i\}$, ($i = 1, 2, \dots, k$) is a predictor of TS Y

is reasonable. Algorithm consists of 3 phases. During the first phase forecast EFT of TS Y , according to (3):

$$\tau_{t+1}^Y = f(\tau_t^Y)$$

therein τ_{t+1}^Y – prognostic fuzzy elementary trend of time series Y ,

τ_t^Y – current fuzzy elementary trend of time series Y ,

f – dependence in fuzzy elementary trends of time series Y .

The second phase involves correction of prognostic fuzzy elementary trend of time series Y in accordance with the components of main trends of the analyzed time series $G\tau_Y$ and TS predictor $G\tau_Z$ respectively:

$$\hat{\tau}_{t+1}^Y = r(\tau_{t+1}^Y, G\tau_Y, G\tau_Z),$$

therein τ_{t+1}^Y – is a prognostic fuzzy elementary trend of time series Y , $\hat{\tau}_{t+1}^Y$ – prognostic fuzzy elementary trend of time series Y after correction, $G\tau_Y$ main fuzzy trend of time series Y , $G\tau_Z$ – main fuzzy trend of time series Z , r – correction rules.

The third phase serves for estimation of prognostic value of numeric time series Y , according to (7).

On this base we suggest the following algorithm for TS.

Algorithm 2.

Step 1. Transformation of numeric TS $Y = \{t_i, x_i\}$, ($i = 1, 2, \dots, n$), into fuzzy TS $\tilde{Y} = \tilde{x}_t$, $\tilde{x} \in \tilde{X}$, $t = 1, 2, \dots, n$:

$$\tilde{x}_i = Fuzzy(x_j), x_j \in X, \tilde{x}_i \in \tilde{X},$$

Here at the intervals where fuzzy sets defined, its form and name are set up by user from object domain characteristics.

Step 2. Transformation of fuzzy TS $\tilde{Y} = \tilde{x}_t$, $\tilde{x} \in \tilde{X}$, $t = 1, 2, \dots, n$ into fuzzy TS of fuzzy elementary trends, is according to (3,4):

$$\begin{aligned} \tau_t^Y &= \langle \tilde{v}_t, \tilde{\alpha}_t, \mu_t \rangle, \\ \tilde{v}_t &= TTend(\tilde{x}_t, \tilde{x}_{t+1}), \tilde{\alpha}_t = RTend(\tilde{x}_t, \tilde{x}_{t+1}), \\ \mu_t &= \min(\mu(\tilde{x}_t), \mu(\tilde{x}_{t+1})). \end{aligned}$$

Beforehand determine a set of FT type names $\tilde{V} = \{\text{Fall, Growth, Stability}\}$, and a set of FT intensity names $\tilde{A} = \{\text{Intense, Average, Weak}\}$.

Step 3. Generation of EFT components change models of TS Y and its forecasting for one period according to (6):

$$\begin{aligned} \tilde{v}_{t+1} &= \tilde{v}_t \times \tilde{v}_{t-1} \times \dots \times \tilde{v}_{t-p} \circ R_{\tilde{v}}(t, t-p), \\ \tilde{\alpha}_{t+1} &= \tilde{\alpha}_t \times \tilde{\alpha}_{t-1} \times \dots \times \tilde{\alpha}_{t-p} \circ R_{\tilde{\alpha}}(t, t-p) \end{aligned}$$

Step 4. Forecast of numeric time series Y with preliminary defuzzification according to formula (7) of FT components

$$\begin{aligned} \tau_{t+1}^Y &= \langle \tilde{v}_{t+1}, \tilde{\alpha}_{t+1}, \mu_{t+1} \rangle \\ x_{t+1} &= x_t + v_{t+1} \cdot \alpha_{t+1}. \end{aligned}$$

Step 5. Application of main trend identification algorithm (see Part 3. Algorithm 1) for TS Y and determination of its components $G\tau_Y = \langle \tilde{v}_{G\tau}^Y, \tilde{\alpha}_{G\tau}^Y, \tilde{\mu}_{G\tau}^Y \rangle$.

Defuzzification of TS Y basic fuzzy trend components is according to (8) and (9).

Step 6. Application of basic trend identification algorithm (see Part 3. Algorithm 1) for time series Z and determination of its components $G\tau_Z = \langle \tilde{v}_{G\tau}^Z, \tilde{\alpha}_{G\tau}^Z, \tilde{\mu}_{G\tau}^Z \rangle$.

Defuzzification of TS Z basic fuzzy trend components is according to (8) and (9).

Step 7. Correction of TS prognostic fuzzy elementary trend Y

$$\begin{aligned} \hat{\tau}_{t+1}^Y &= r(\tau_{t+1}^Y, G\tau_Y, G\tau_Z): \\ \hat{\tau}_{t+1}^Y &= v_{t+1} \cdot \alpha_{t+1} + v_{G\tau}^Y \cdot \alpha_{G\tau}^Y + v_{G\tau}^Z \cdot \alpha_{G\tau}^Z \end{aligned}$$

Step 8. Calculation of corrected prognostic value of numeric TS Y for a one period

$$x'_{t+1} = x_t + \hat{t}_{t+1}^Y.$$

The proposed approach to forecast was tested for short-term forecast of IT companies sales of Ulyanovsk region in Russia (time series Y consists of 12 values), the predictor was expertly selected time series Z as an average number of employees of IT companies engaged in production of new software.

Table I shows the results obtained estimates predict.

Table I. Evaluation of forecasting

Evaluation	Song model[8]	Fuzzy Tend[9]	Proposed model
MSE	0,025	0,202	0.0123

To check the obtained forecast values we used the criterion MSE:

$$MSE = \frac{1}{n} \sum_{t=1}^n (x_t - x'_t)^2$$

The results of the experiment demonstrates that the suggested approach implementing modified method of EFT forecasting might be used for short-term forecasting of time series when there is an expert assumption about existence of predictor time series.

V. CONCLUSION

This article proposes the approach to the analysis of the software market trends in technologies, scientific achievements and user requirement, expressed in time series. The proposed approach allows to uncover ways to create a new useful ideas for software development. We propose the algorithm expanding the opportunities of TS forecasting on the base of fuzzy trends, as the historical software TS are of small length. The experiments carried out demonstrate the functionality and increase of forecast accuracy when applying the suggested algorithm.

The advantages of the proposed approach:

- The results of forest consider the results of the analyzed time series main trend.
- It does not demand for highly qualified users.
- The analyzed time series might be of short length.
- The analyzed time series might be of various length.

The future research will be connected with involving of time series similarity coefficient into forecasting algorithm.

ACKNOWLEDGMENT

This work has been partially funded by the projects no. 13-01-00324 and no. 14-07-00247 of the Russian Foundation for Basic Research.

REFERENCES

[1]. Holt C.C. Forecasting trends and seasonals by exponentially weighted moving averages // O.N.R. Memorandum, Carnegie Inst. of Technology. 1957. № 2.

[2]. Kendall, M. Time series. Translated from English, Yu. P. Lukashina. – Moscow: Finansy I statistika, 1981. 199 p. (rus)

[3]. Anderson T. W. Statistical Analysis of Time Series.. New York: John Wiley and Sons, Inc., 1971.

[4]. Box, J. Time series analysis: Forecasting and control: Translated from English; ed. by F. Pisarenko – Moscow: Mir, 1974. – 406 p (rus).

[5]. Zadeh, A. Lotfi. Fuzzy Sets / Lotfi A. Zadeh // Information and Control. – 1965.

[6]. Sabic, D.A. Evaluation on fuzzy linear regression models / D. A. Sabic, W. Pedrycz // Fuzzy Sets and Systems. 1991. №23. P. 51-63.

[7]. Chen, S. M. Forecasting enrollments based on high-order fuzzy time series/ S.M. Chen // Cybernetics and Systems: An International Journal. 2002. – № 33. P. 1–16.

[8]. Song, Q. Fuzzy time series and its models / Q. Song, B. Chissom // Fuzzy Sets and Systems. 1993. № 54.- P. 269–277.

[9]. Afanaseva T. V., Yarushkina N. G. Fuzzy Time series with fuzzy tendency. In Vestnik Rostovskogo Gosudarstvennogo Universiteta Putey Soobzheniya' (Vestnik RGUPS) Rostov-on-Don, Russia, 2011.- P. 7-16.(rus)

[10]. Perfilieva, I. et al. Relaxed Discrete F-Transform and its Application to the Time Series Analysis / I Perfilieva, N Yarushkina, T Afanaseva // Da Ruanetal (Eds.): Computational Intelligence. Foundations and Applications (Proc.of the 9th Int. FLINS Conf.), pp. 249 --255, World Scientific, Emei, Chengdu, China, 2-4 August, 2010.

[11]. Tanaka, H. Linear regression analysis with fuzzy model / H. Tanaka, S. Uejima, K. Asai // IEEE Transactions on Systems, Man and Cybernetics. 1982. № 12(6). P. 903–907.

[12]. Yarushkina N., et al. Time Series Processing and Forecasting using Soft Computing Tools. - Lecture Notes in Computer Science, Vol. 6743, Proceedings of 13th International Conf. RSFDGrC-2011. Springer-Verlag, Berlin Heidelberg, 2011, XIII.-p. 155-163.

[13]. Perfilieva, I. et al. Soft computing tools for time series analysis and forecast/ I. Perfilieva, N. Yarushkina, T. Afanasieva, A. Igonin, A. Romanov, V. Shishkina Proceedings of the 9th Int. Conf. on Application of Fuzzy Systems and Soft Computing (ICAFS 2010) Eds. R. A. Aliev, K. W. Bonfig, M. Jamshidi, W. Pedrycz, I.B. Turksen, Prague, August 26-27, 2010, VERLAG b- Quadrat Verlag, pp. 50—60.

[14]. Gregory, Allan W.; Hansen, Bruce E. (1996). Residual-based tests for cointegration in models with regime shifts. Journal of Econometrics 70 (1): 99–126.

[15]. Engle, Robert F.; Granger, Clive W. J. (1987). Co-integration and error correction: Representation, estimation and testing. Econometrica 55 (2): 251–276. .

[16]. Granger, Clive. (1981) Some Properties of Time Series Data and Their Use in Econometric Model Specification. Journal of Econometrics 16 (1): 121–130.

[17]. Yarushkina N.G. Osnovy teorii nechetkikh I gibridnykh sistem [The fundamentals of fuzzy and hybrid systems theory]: tutorial / N.G. Yarushkina. – Moscow: Finansy i statistika, 2004. 320 p. (rus)

[18]. Tsenga, F. M. Fuzzy ARIMA model for forecasting the foreign exchange market / F. M. Tsenga, G. H. Tzengb, H. C. Hsiao-Cheng Yua // Fuzzy Sets and Systems. 2001. №118.

[19]. Pedrycz W., Chen S.M. (Eds). Time Series Analysis, Modeling and Applications: A Computational Intelligence Perspective (e-book Google). – Springer-Verlag, Berlin Heidelberg, 2013.- (Intelligent Systems Reference Library, Vol. 47). 404 pp.

Detecting and highlighting text in images

Ivan Pakhomov

Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
ivan_pahomov@mail.ru

Abstract— The work describes the problem of detecting and highlighting text in images. For the comparison, it contains the existing methods to solve this problem, advantages and disadvantages of them and, as a result, own approach to solving the task is proposed.

Keywords—text detection; images; recognition; classifiers; classification features.

I. INTRODUCTION

Optical text recognition of the images is a very important issue, which has significant amount of practical applications: indexing photos and videos, mobile text recognition, robot navigation.

Nowadays, a digital camera is available in almost all modern phone, smartphone and tablet. The number of digital photos and videos on the Internet is increasing significantly. According to official statistics [1], 100 hours of video are uploaded to video hosting service YouTube every minute. Thus, there is a need to find a way to effectively manage these multimedia resources and analyze their contents.

The text which contains high – level semantic information is well suited to solve the problem. For example, the text contained in images on the Internet often relates to the content of web pages. The text on the covers of books and magazines is often necessary for indexing: two books with identical design but different titles will look the same if the text on the cover is unknown. News headlines as well as subtitles usually contain information about when and where the event occurred. Moreover, in contrast to other information that can be obtained from images, text is created by people, so it can directly determine the contents without any calculations.

In the modern world, people are surrounded by a vast amount of textual information such as labels, signs and billboards. Unfortunately, not everyone has the opportunity to use it. For example, a device that reads aloud can be useful for the visually impaired. However, healthy people may also face challenges, e.g. the problem of the language barrier in a foreign country. For that category of people, there are programs which can translate text from photos to the language selected by the user.

Navigating using GPS or Glonass is convenient enough, but it has some disadvantages. For example, in the places where the satellite signal is not available, it cannot be used. That's why, in case of an emergency, rescue robots need to use visual information for orientation. House numbers, signs on the buildings, road signs, various schemes – all of this can be

used, but only if robot can recognize the text on them. In addition, there are many practical problems where it is necessary to be able to automatically recognize the text in the image: scan car numbers for automatic fixing of violations or mapping various organizations using panoramic images of streets.

In some cases, text selection has independent meaning. For example, time of appearance of the title in the video news shows the beginning of a new scene that can be used in automatic video summarization. Or maybe there is a need to attract the user's attention to some text.

K. Jung and colleagues [2] gave the definition of a system for obtaining information from a text image, which consists of four steps:

- *Detection*. In this step, it is determined whether there is text in the image or not.
- *Localization*. At the second stage, the location of the text is determined. Usually, the result of this step is a rectangle, which contains text.
- *Extraction*. Highlighted text areas are cleared of everything extraneous, background is removed. The text is grouped into words and symbols.
- *Recognition*. At the last stage, there is a transformation graphics to text.

Out of all stages, detection and localization of text are critical to overall system performance. Moreover, these two steps may be considered together. Finally, if the text is found, its location is determined.

In recent years, there have been suggested large number of methods to solve these problems, but quick and accurate selection of text in the photos is still a significant problem because of the large diversity of fonts, sizes, colors, methods of spatial orientation. Often, the problem is exacerbated by complex background, lighting changes, obstacles, image distortion and loss of quality in compression.

In this paper, the task is to explore some of the currently existing methods for the detection of text and to build own system to detect and highlight text in the images of poor quality.

II. RELATED WORKS

The existing methods for highlighting text can be divided into two groups, based on the analysis of regions and on the analysis of connected components. Methods, based on the analysis of regions, perform texture analysis of the image fragments. The vector of values, consisting of numerical

evaluations of different textural properties, is generated for each fragment, called region. This vector is input to the input classifier, which estimates the degree "text" of the region. Then, adjacent text regions are combined to produce blocks of text. From the fact that textural features of text areas differ from the signs of non-text areas, such methods can detect text even in noisy images.

Methods based on the analysis of connected components divide the whole image into separate components of any kind, for example, by color. Non-text components are discarded by means of heuristics or classifiers. Because the number of found segments is relatively small, these methods have a lower computational cost, than methods based on the analysis of regions, and the selected text components can be used directly for recognition.

Although the existing methods claim impressive results, there are several problems. The methods based on the analysis of regions are relatively slow, and their performance is sensitive to the text's location. On the other hand, methods based on analysis of connected components cannot accurately segment text without information about the location and scale of the text. Moreover, there are many non-text components that are easily confused with the text in individual analysis. For example, wheel of car can be taken for the letter "O". Some works offer mixed approaches which use methods based on the analysis of regions and analysis of connected components.

A. Analysis of regions

The system described in the paper [3] is an illustrative example of the system based on the analysis of regions. Adam Coates et al proposed an interesting approach based on learning without a teacher to receive signs. They developed the system, which consists of three stages:

- Unsupervised learning algorithm. It is used to get a set of calculated features of image fragments, obtained from a training set;
- The number of attributes is reduced by the use of spatial association [4];
- The classifier is trained to select text.

At the first stage of the system, image fragments collection of 8 by 8 pixel grayscale is assembled. All fragments are pretreated. For this purpose, the intensity values and gradient of each fragment are normalized, by subtracting the values of these variables at each point of mathematical expectation and by multiplying the resulting difference by the standard deviation. Then, whitening [5] [6] based on the analysis of zero components is applied. Whitening is used for pre-processing of the vector's features. It normalizes the values of the vectors, so that the coefficients of variation of the individual values are equal. The authors use square fragments side of 32 pixels. For each area of 8 by 8 pixels in this fragment feature vector is calculated. After that, averaging association is used, that is just a new vector is calculated as the arithmetic average of all the vectors, describing eight-pixel area. If this is not done, the amount of information will increase many times.

There is using a sliding window method for detection. The vector of signs is calculated for each thirty-second pixel image fragment. These calculations are performed for differently scaled images to detect text of different sizes. Then, each pixel of the original image is assigned with the maximum result of the classifier, got for all fragments at different scales. The obtained values are binarized with a certain threshold, and the result is a mask which indicates the presence of the text in the image. By varying the threshold of binarization, it can obtain different values of accuracy and completeness. Accuracy is calculated as the ratio of correctly labeled pixels to the total number of labeled pixels, completeness – as the ratio of the number of correctly labeled pixels to the total number of pixels that had to be mark.

This approach does not show the highest results (accuracy of 61% at a density of 69%), but is interesting, because it does not use some logically justified signs and gets them itself.

B. Analysis of connected components

In the paper [7], B. Epstein, E. Ofek, and E. Wexler presented the allocation method of independent components based on the operator of stroke width. The operator of stroke width – a statement that assigns each pixel of the original image to stroke width which it is most likely treated.

First, the verges are allocated using the Sobel operator. Then, starting from a point on the selected face towards increasing gradient for light text on a dark background, or decreasing, for dark on light, all the pixels to the next face are marked. For all the marked pixels, their stroke width equal to their number is indicated. If a pixel has been previously marked, the value of its stroke width varies insignificantly. Neighboring pixels, for which the stroke widths differ by no more than three times, are together grouped, and connected components are formed. Next, the filtered components are grouped into rows. At this stage, the components are filtered further. For example, individual components are not considered. Text within a line should have approximately the same stroke width, the distance between characters, character size. Additionally, the average color of adjacent characters is compared.

The results stated in the article [7] are very high: 71% accuracy at a density of 60%. It is important to emphasize, that this algorithm is quite fast, 15 times faster than the closest analogue. It is unknown how and on what machines its performance was tested. Also, due to the fact that the connected components have already been allocated, it does not need additional text extraction. As disadvantages, should be noted that, all the freestanding symbols and non-horizontal lines are discarded when filtering. This circumstance greatly reduces the completeness.

C. Kumar and A. Perrault [8] implemented the algorithm described in the previous article [7] in the Nokia mobile phone N900. In the process of implementation, they encountered some difficulties. For example, they could not simultaneously find light text on a dark background and dark on light: for this they needed to run a search twice. In addition, they had to change or omit some filtering rules components, due to poor

results. In the course of such improvements, the results on a specially selected set of data: 99% of accuracy, 100% of completeness.

The disadvantages of this method can be seen from most of its description. Like all methods based on analysis of the connected components, this approach filters out a single character. Like in most methods, it is impossible to distinguish handwritten text and text with merging characters, because it will stand out as one component and discarded during filtration.

C. Hybrid method

The hybrid method is an attempt to combine the analysis of independent components and the analysis of regions to highlight text. Thus, in [9] a system consisting of three stages is proposed. In the first pretreatment step, an analysis region is used to find regions which can contain text. The gradient direction histogram is used as a feature, and the cascade classifier is used as a classifier. It should be noted that the purpose of this step is not an accurate selection of the text, but the definition of the probability that in a given area may be text. To do this, for each piece of image, regardless of whether it is accepted or rejected, we translate the output values of the classifier in the posterior probability using a calibration method of the classifier. Thus, probability maps are constructed for each image in the pyramid, and then they are projected to the original image and create a map of probabilities for the original image. This card is used to make adaptive binarization by Niblack adapted algorithm, in which each component is assigned a certain intensity value. The next step is a transition to the independent components analysis. To do this, we consider only the components with certain intensity values.

To analyze the components the conditional random field model is used with the following features:

- Normalized width and height.
- The ratio of width to height.
- The ratio of the number of pixels belonging to the component number of pixels belonging to the minimum bounding rectangle.
- The average value of the probability that the pixels inside the components are part of the text. For this feature, there is using the map of probabilities, obtained in the analysis phase regions.
- Compact – the ratio between the area of a rectangle describing the square perimeter and components. It allows filtering out components having a complex shape.
- The average value of the gradient at the boundary components.

From the analysis of signs, it can be clearly seen that this approach allows us to find the rows of any shape, but filters out-standing characters. The results of testing the system described: 67% accuracy, completeness 69%. The speed of this method is far superior to methods based on the analysis of

regions, but inferior to the methods based on analysis of components.

III. PROPOSED APPROACH

Based on the research, following way to solve the problem is proposed as the improvement: to build a classifier stage by stage (by cascade) using boosting algorithm to enhance the weak classifiers (decision trees).

Boosting algorithms such as discrete AdaBoost, real AdaBoost, LogitBoost, Gentle AdaBoost, are used in pattern recognition problems, because they are poorly exposed retraining, compared with other machine learning algorithms. Moreover, the use of boosting allows easy use of the feature vector of high dimensionality.

All boosting algorithms are close in their structure, so therefore from now will be considered real AdaBoost, which, according to the results presented in this paper [10] is the best of all algorithms for boosting two-class classification.

A. AdaBoost

Real AdaBoost classifier [11] is a generalization of discrete AdaBoost [12]. Consider both of them.

Suppose we have training set $(x_1, y_1), \dots, (x_N, y_N)$, where x_i – vector of features, and $y_i \in -1, 1$ – class label. Then the classifier:

$$F(x) = \sum_{m=1}^M c_m f_m(x),$$

where $f_m(x)$ – weak classifier, which returns a value from the set $-1, 1$, and c_m – constants. The result of the classifier is defined as the sign of the function $F(x)$:

$$\text{sign}(F(x))$$

AdaBoost trains weak classifiers on a weighted training set, increasing the weight of the elements that were incorrectly classified. These steps are repeated for all the weighted values, and then the final classifier is represented as a linear combination of the classifiers received for each of the steps. Below is a detailed algorithm of discrete AdaBoost:

1. Initial weights are specified: $w_i = \frac{1}{N}, i = 1 \dots N$
2. For each $m = 1, 2, \dots, N$
 - a. Train classifier $f_m(x) \in -1, 1$ using a weighted training set.
 - b. Calculate the classification error
$$\text{err}_m = E_w \left(1_{(y \neq f_m(x))} \right), c_m = \log \left(\frac{1 - \text{err}_m}{\text{err}_m} \right),$$

$$E_w$$
 – the mathematical expectation calculated for the weighted training set, and $1_{(s)}$ – identifier of the set S .
 - c. Refresh weights
$$w_i = w_i \exp \left[c_m 1_{(y \neq f_m(x))} \right],$$

$i = 1, 2, \dots, N$ and normalize them, so that $\sum_i w_i = 1$, $S_w = \sum_{i=1}^N w_i$, $w_i = \frac{w_i}{S_w}$

At each iteration, the weights of misclassified vectors are increased by the value determined by the classification weighted error.

d. The result: $sign(\sum_{m=1}^M c_m f_m(x))$

L. Breiman [13] demonstrated that the use of classifiers based on trees as weak classifiers gives good results. Besides, various tests have shown that increasing the number of classifiers increases the accuracy of the algorithm on the test sets, which demonstrates the stability of AdaBoost to overfitting.

Real AdaBoost is a generalization of discrete AdaBoost. It uses predictors that return the probability of belonging to the class. The set of values of the weak classifier is already in the field of real numbers. $sign(f_m(x))$ – defines a class and $|f_m(x)|$ – the probability. Below is the algorithm of real AdaBoost:

1. Initial weights are specified: $w_i = \frac{1}{N}$, $i = 1 \dots N$
2. For each $m = 1, 2, \dots, N$
 - a. $p_m(x) = P_w(y = 1|x) \in [0, 1]$
 - b. $f_m(x) \leftarrow \frac{1}{2} \log \left(\frac{p_m(x)}{1 - p_m(x)} \right) \in \mathbb{R}$
 - c. Refresh weights $w_i = w_i \exp[-y_i f_m(x_i)]$
 $i = 1, 2, \dots, N$ and normalize them, so that $\sum_i w_i = 1$, $S_w = \sum_{i=1}^N w_i$, $w_i = \frac{w_i}{S_w}$
 - d. The result: $sign(\sum_{m=1}^M f_m(x))$

To reduce the computation time for these models without a significant loss in accuracy technique of circumcission is used. In the course of the algorithm, while the number of trees is increasing, large number of examples from the training set is classified correctly. Therefore, weight of these examples is decreased. Examples with low weights give a small contribution to training of the weak classifiers. Therefore, these examples may be removed at training, without a lot of harm to the learning outcomes of the weak classifier. For this purpose the threshold value may be specified for cutting the training set. It should be noted that this procedure is repeated for each weak classifier, and these clipped examples can be used for training in the following stages.

B. Decision trees

Decision tree – is a balanced binary tree, which can be used for classification and for regression tasks. The procedure of the prediction begins with the root vertex. From the each non-leaf vertex prediction procedure goes to the left or right, depending on the value of the specified variable feature vector, whose index is stored in the current node. The value of the variable is compared with a threshold value stored in the node. If the value

is less than the threshold, the procedure goes to the left, otherwise – to the right. Each node uses a pair of index and variable threshold. This pair is called a partition. When it reaches a leaf node, the value stored in it is used as a result of the classifier. Trees are built recursively starting from the root node. All the training set is used to split root. At each node, the best split is chosen using some criterion. In machine learning classification Gini impurity is used [13]. This criterion is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset. To compute Gini impurity for a set of items, suppose i takes on values in $\{1, 2, \dots, m\}$, and let f_i be the fraction of items labeled with value i in the set:

$$I_G(f) = \sum_{i=1}^m f_i(1 - f_i) = \sum_{i=1}^m (f_i - f_i^2) = 1 - \sum_{i=1}^m f_i^2$$

All data are separated in accordance with the selected partition into two subsets, which are used for training of the left and right subtrees. Recursive procedure can stop at one of the following cases:

- maximum depth of the tree is reached;
- power training set in the node is less than a given threshold, and is not representative;
- all examples in the node belong to the same set, or, in the case of the regression, the variation between them is small;
- the best selected separation does not give significant gain in comparison with a random choice.

C. Viola–Johns classifier

P. Viola and M. Jones [14] presented an interesting approach to the construction of classifiers, which they used to find human faces in the image. This idea is to build a cascade of multiple classifiers, and discard individual fragments gradually, at each step.

It is well known that more complex classification function is more accurate, but its ability to generalize is smaller. Minimization of structural risks provides a formal method for selecting classifier with the right balance of complexity and correctness, in the case, where the main factor is to reduce the errors.

Another important limitation is the computational complexity. The computation time and error – completely different things, that's why it is theoretically impossible to choose an optimal balance. Nevertheless, for many classification functions computation time depends on the structural complexity. In such cases, the cost reduction of time depends on the complexity reduction. Nevertheless, the computation time depends on the structural complexity for many classification functions. . In such cases, the reduction of time cost depends on the reducing the complexity.

But this direct analogy does not work in applications where the balance between the marks strongly shifted towards any class. For example, when you select text, there are thousands of

fragments without the text and only a small part contains the text. Oddly enough, there are good results can be achieved with a sufficiently high percentage of correctly classified fragments and very fast classification.

The key point is that despite the fact that it is impossible to build a simple classifier with a very low error rate, but, in some cases, it is possible to build a classifier, which is practically never wrong to classify text areas.

For example, it is possible to create a very fast classifier, which is correct in the text fields, but wrong in 90% of cases on the non-text areas. Such a detector can be used for pre-filtering: if the fragment is marked as "no text", it immediately discarded, and if the fragment is marked as "text", it requires an additional classification. Sequence of such classifiers (each following is harder, slower and more accurate than the last) is a cascade of classifiers.

For classification, slightly modified AdaBoost can be used. Normal AdaBoost tries to minimize the total error at each step, but we want to minimize the number of misclassified text regions. The idea is to change the balance of the scales in favor of the data with positive marks. But this is not enough just to change the balance in the first step, because AdaBoost weights will be changed and the second weak classifier will be symmetric. Therefore, it is necessary in addition to the initial balance variation, to change the standard AdaBoost formula:

$$w_i = w_i \exp\left[\frac{1}{N} y_i \log(\sqrt{k}) - y_i f_m(x_i)\right]$$

$i = 1, 2, \dots, N$, where k – positive versus negative preference coefficient.

IV. FEATURES FOR CLASSIFICATION

In the classification a large number of different features is used. There are ten the most commonly used:

- The standard deviation of intensity

If the value of the intensity varies very slightly within the limits image fragment, it is likely that this fragment does not contain any text.
- The standard deviation of the gradient

This criterion shows the range of values of the gradient within the fragment. Can filter out solid areas, but requires additional calculation of the gradient values.
- Entropy

Can be calculated by the formula:

$$H(x) = - \sum_{i=1}^n p(i) \log_2 p(i)$$
- Statistics of derivative by X

This feature uses the observations described in the article [9], consisting of the different values of the derivative in different regions containing the text

- Statistics of derivative by Y

Is identical to statistics of derivative by X
- Histogram of gradient values

Before calculating characteristic values, original image fragment is always scaled to the size of 20 by 20 pixels, thereby obviating the need for normalization values of the histogram. Values divided into twenty equal intervals, whereby this feature gives us the twenty values.
- Histogram of the intensity values

Is identical to the histogram of gradient values
- Histogram of intensity gradient

This histogram is built on two values. All quantities that are included in one interval of intensity histogram are divided into intervals by value gradient Splitting happens for twenty intervals. In total, the result is four hundreds values.
- Histogram of bar width

The idea of this feature is that the bar width distribution in text areas are different from the bar width distribution in non-text areas. Before calculating, the image is compressed to a size of 20 by 20 pixels, to increase the speed of operation, because the transform is a time-consuming operation. All values are distributed, as in other characters, using histograms, at the twenty slots.
- Separation of Gaussians

This feature shows how many divided the expectations of Gaussians. If they almost merge with each other, then the background and text are not distinguishable. Can be calculated by the formula:

$$\Delta M = \frac{|M(G_1) - M(G_2)|}{255},$$

where $M(G_1)$ – the mathematical expectation of the Gaussian;

255 – the maximum difference in the intensities of eight-bit image.

Thus, using the features listed above, as well as combinations thereof, different results can be obtained, as different speed and precision. For example, we may proceed from operation speed characteristics: at the first cascades we can use faster but less accurate features and slower but more accurate on the latter.

V. LOW QUALITY IMAGES

There is pretreatment required, for the images with low resolution and quality to remove noise, to smooth background and to enhance the contrast between text and background. For this purpose, Weiner filter [15] is suitable. Also, in this case, the bilateral filter is appropriate to use. Bilateral filter allows to enhance the contrast between areas of different colors and to make transitions sharper. The latter is particularly important in cases where the text is almost completely blends into the background.

For example, in figure 1 graph of the results of the classifier on the worst images of the bilateral filter parameters: radius and range of weight. It can be seen that the harmonic mean varies widely and can exceed eight times the results obtained without pre-filtering.

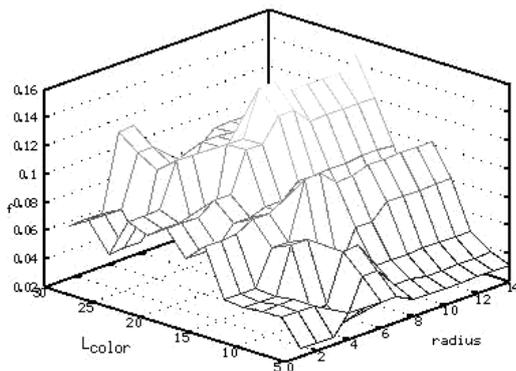


Fig. 1. Use of bilateral filter

VI. RESULTS

The system was developed using C++ programming language. It was trained and tested on ICDAR 2013 data sets. Testing was carried out according to the procedure, described in the article [3]. All fragments of the image, which were classified and labeled as text, formed a mask. For the true values markup dataset was used. Further, completeness and accuracy were calculated using the following formulas:

$$Accuracy = \frac{N_{true+}}{N_+}$$

$$Completeness = \frac{N_{true+}}{N_{true+} - N_{true-} + N_-}$$

where: N_+ – the number of pixels classified as text;
 N_- – the number of pixels classified as non-text;
 N_{true+} – the number of text pixels classified as text;
 N_{true-} – the number of non-text pixels classified as – non-text.

The results of the classifier for now: accuracy – 51% completeness – 88%. Figure 2 shows the image on which the classifier showed the best result. This image characterized by very good contrast between text and background, weak blurring and it is easily readable.



Fig. 2. Image with the best classifier's result

Figure 3 presents image on which the classifier showed the worst results. The text on this image almost merges with the background, and the stones, that stood out as the text, look much clearer.



Fig. 3. Image with the worst classifier's result

Of course, there is still much work to optimize a system highlighting and detecting text, but the most importantly, the valuable theoretical basis has been laid, and, on this basis, it is possible to construct an effective and power system to solve not only this problem, but related problems too.

CONCLUSIONS AND FURTHER WORK

The problem of this paper is considered upon the problem of detecting and highlighting text in images. Its importance in today's multimedia world was emphasized and the possibility of its application was described. Further, the most known and widely used approaches to solving the problem were considered as well as their pros and cons, and possible update options. On this basis, the more optimal approach was offered and the nuances of the implementation were discussed as well as results. As already mentioned above, the proposed system has a great list of practical applications, but immediate plans include the solution of graphic content indexing problem in the video, and, possibly, porting it to the mobile device to create a mobile application for the recognition of text from the photos taken on the device.

REFERENCES

- [1] YouTube Press (2014). *YouTube Official Statistics*. Retrieved April 1, 2014, from <http://www.youtube.com/yt/press/statistics.html>
- [2] Jung K., Kim K., Jain Anil K. Text information extraction in images and video: a survey. *Pattern Recognition*, 2004, 37(5), 977–997.
- [3] Coates A., Carpenter B., Sathesh S. Text Detection and Character Recognition in Scene Images with Unsupervised Feature Learning.

- Document Analysis and Recognition, *ICDAR 2011 International Conference*, 2011, 440-445.
- [4] Boureau Y.L., Bach F., LeCun, Y. Learning mid-level features for recognition. *Computer Vision and Pattern Recognition (CVPR)*, 2010 IEEE Conference, 2559–2566.
- [5] Hyvarinen A., Oja E. Independent component analysis: algorithms and applications. *Neural Networks Oxford*, 2000, 13(4-5), 411– 430.
- [6] Picard Rosalind W. *Decorrelating and then Whitening data*. Retrieved December 25, 2013, from <http://courses.media.mit.edu/2010fall/mas622j/whiten.pdf>
- [7] Epshtein B., Eyal O., Yonatan W. *Detecting Text in Natural Scenes with Stroke Width Transform*. Retrieved December 20, 2013, from http://www.math.tau.ac.il/~turkel/imagepapers/text_detection.pdf
- [8] Kumar S., Perrault A. *Text Detection on Nokia N900 Using Stroke Width Transform*. Retrieved November 19, 2013, from http://www.cs.cornell.edu/courses/cs4670/2010fa/projects/final/results/group_of_arp86_sk2357/Writeup.pdf
- [9] Yi-Feng P., Xinwen H., Cheng-Lin L. A Hybrid Approach to Detect and Localize Texts in Natural Scene Images, 2011, *NJ, USA: Trans. Img. Proc. Piscataway*.
- [10] Friedman Jerome, Hastie Trevor, Tibshirani Robert. Additive Logistic Regression: a Statistical View of Boosting. *Annals of Statistics*. 1998. T. 28. c. 2000.
- [11] Schapire Robert E., Singer Yoram. Improved Boosting Algorithms Using Confidence-rated Predictions. *Machine Learning*. 1999. T. 37, № 3. C. 297– 336.
- [12] Freund Yoav, Schapire Robert E. Experiments with a New Boosting Algorithm. *ICML. Morgan Kaufmann*, 1996. C. 148– 156
- [13] Breiman, L., Friedman, J. H. Classification and regression trees. *Wadsworth Publishing Company*, 1984.
- [14] Viola Paul, Jones Michael. Fast and Robust Classification using Asymmetric AdaBoost and a Detector Cascade. *Advances in Neural Information Processing System 14. MIT Press*, 2001. C. 1311–1318.
- [15] Wiener Norbert. Extrapolation, Interpolation, and Smoothing of Stationary Time Series. *The MIT Press*, 1964.

Using multidimensional ontology of electronic document for solving semantic indexing problem

Lanin V.

Department of Business Informatics
National Research University Higher School of Economics
Perm, Russia
lanin@perm.ru; mistika93@mail.ru

Sokolov G.

Computer science department
Perm State National Research University
Perm, Russia
sokolovgeorge@gmail.com

Abstract – The paper describes an approach to semantic indexing of electronic documents based on ontology that describes the structure, type of document and its contents. In addition, existing ontology descriptions of documents are considered and the differences between the proposed multidimensional ontology from them are described. The solution of the problem of analysis of administrative regulations is described as an application of the approach. An algorithm for implementing semantic indexing based on multi-agent paradigm is proposed.

Keywords – multidimensional ontology, semantic indexing, intellectual agents

I. INTRODUCTION

Transition from processing structured data to unstructured data processing is observed in modern information systems. New classes of systems, such as social networking, corporate portals, wiki-resources, etc. became an integral part of the information process. The key point of such systems is "content", which concept can be generalized to "electronic document." Unstructured nature of information raises the question of the transition from traditional indexing documents based on unrelated keywords («bag of words») to the so-called semantic (conceptual) indexing. Semantic (conceptual) document indexing is an indexing, in which synonyms are reduced to the same concept, and disambiguation are separated into different conceptual units [3].

Semantic index of document can become the basis for solving many problems in the processing of electronic documents, in particular, their search, analysis and classification, cataloging and efficient storage, generation and support their life cycle. It's needed to have consolidated knowledge about their structure and content.

Base of semantic index is ontological resource in that following information about the following aspects of electronic documents is needed: electronic document format; type of electronic document; the structure of an electronic document.

When ontological resource is created, it includes concepts related to all three aforementioned aspects of a document information representation. Each of them is described by ontology. Concepts of the various aspects have to be linked. Thus, a single ontology of electronic documents is being

created. In addition, the resource should support the ability to expand and specify the settings on the solution of specific problems arising in the processing of documents in a variety of information systems throughout their life cycle.

Thus, in the paper existing ontology resources for describing documents will be examined, an approach to the description of multidimensional ontology will be proposed and an algorithm for semantic indexing based on multi-agent approach will be provided.

II. EXISTING DOCUMENT ONTOLOGY

Dublin core [4] – is a set of metadata used to describe documents of various types (publications, audio records, video records). This set specification has status of official international standard (ISO: 15836 2003). The standard has two levels: Simple, comprising 15 elements and Qualified having three additional elements and element refinements (or qualifiers), which refine semantics of the elements. The main feature of Dublin Core is that every element is optional and might be repeated. Dublin Core is a powerful instrument used to describe resources of various types. The fact that it is widespread and flexible is its overwhelming advantage. However, it describes documents tags, i.e. information having indirect correlation with the document content. In this case it is impossible to describe other aspects of the electronic document.

Project ontologies «docOnto» [3] developed by German research group KWARC (Knowledge Adaptation and Reasoning for Content) differ from other projects oriented on formal structure description development (CNXML document ontology) and document semantics (OMDoc document ontology). Members of this group also develop mechanisms of semantic document indexing and tools for document processing. CNXML document ontology (Connexions Markup Language) describes such terms as paragraph, section, reference etc. Ontology is formalized on UML. It gives detailed description of the document. Unfortunately, work in this direction is frozen, last changes date back 2007. One more direction in document ontologies creation is semantics description of documents for narrow subjects, where documents are well formalized, for example mathematical OMDoc documents. Mathematical Terms, theorems and several other terms are included in ontology.

Document ontology SHOE [5] describes most types of documents. Academic papers are given particular emphasis. Dublin Core reference books and Document Classifier PubMed were the resource.

Document Ontology of Research Centre Linked Data DERI is developed by scholars of Irish Institute DERI (Digital Enterprise Research Institute) and is described in RDFS and OWL-DL [9]. Terms referring project activity documentation are given in the ontology. Developers purposefully refused modelling structure and document content to accommodate flexibility and interoperability.

Muninn project document ontology became the result of processing archive documents of the First World War within the project Muninn WW1 [7]. The Ontology describes bibliography, origins and storage description of the digital item. Most ontology classes are child classes of FOAF. That decision was compatibility possible, on the other hand, make adding additional features of document processing possible, i.e. features for representation document pages, copyright description, etc. One of the main ontology classes is Document, which is integrate class of FOAF Document and Creative Commons Works. Page class describes document pages, in its turn, Image class describes digital page image. Description of different document aspects, document structure in particular, is a significant benefit of this ontology. However, structure description is initially oriented on digital images of archive documents.

Each listed above document ontology has its advantages and disadvantages. We create own ontology specialized on academic paper description.

III. USING SAMPLE

Consider the example of the proposed approach based on the work with electronic administrative regulations (EAR) [9]. The basic approach to the development of software tools to support the EAR conduct is ontological modeling. Used in the process ontologies are placed in multi-level repository [10], which contains the domain ontology and ontology normative-reference documents. Domain ontology defines the terms used in the documents, namely it describes concepts such as "process", "operation", "artist", etc., in addition, there are included the various classifiers. Ontology of normative-reference documents, in particular, the ontology of the regulation (Fig. 1) describe the structure of the characteristic elements of documents.

As a result of text description analysis (decomposition) will be built a conceptual model of regulation that, first, to allow it to verify (check structure, identify duplication of information, etc.), and secondly, will link the fragments of a text document with the relevant concepts of the ontology. In addition, the conceptual model of documents could be used to set the "semantic" relationships between different documents and visualization of these links.

Next, consider how ontologies are used in multi-agent semantic indexing algorithm. Domain ontologies used at semantic analysis step. Ontologies that describe the structure of the document (for example, the aforementioned ontology of

regulatory-reference documents) are used at the stage of structural analysis. All ontological resources described in RDF-format. Consider in more detail the steps of the analysis of documents used in the algorithm based on semantic indexing agents (fig. 1).

IV. MULTI-AGENT SEMANTIC INDEXING ALGORITHM

A. Document analyses steps

Simplifying the problem we assume that first step of text analysis process was made (for instance using Yandex Mystem[11]), i.e. a set of morphological descriptors for each word have been obtained. All others steps are performed by agent-based semantic indexing. As it could be seen on fig. 1 syntax analysis is not used because it has high time complexity. Instead of this words order in sentence is considered.

Next step is a semantic analysis. The result of the semantic analysis is a semantic descriptor of plain text that binds the morphological descriptors to the elements of the domain ontology. Stop words are skipped.

Next step is a structural analysis. The structural analysis uses document's structure, ontology that describes structure and semantic descriptors of plain text. At this step every concept of structural ontology tries to binds to corresponding structural document element. The result of structural analysis is semantic descriptor of whole text.

Descriptors (morphological, semantic) are a set of tags, which marks each words in the text.

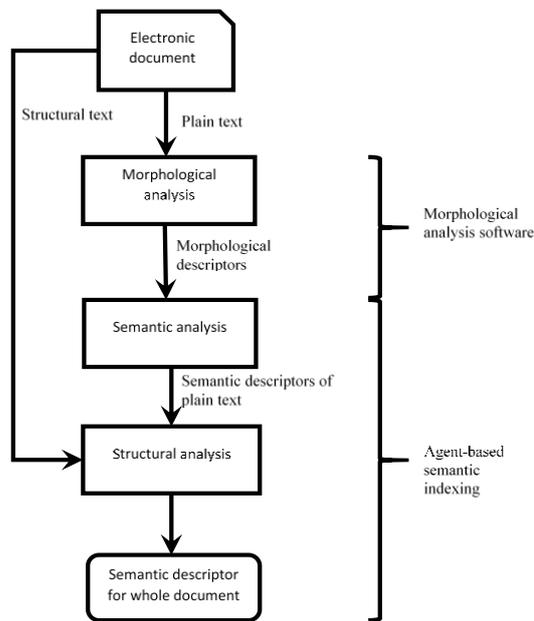


Fig. 1. Steps of document analyses

B. Agent-based solution

Further let us consider the process of building a semantic index based on multi-agent approach (see Fig. 2).

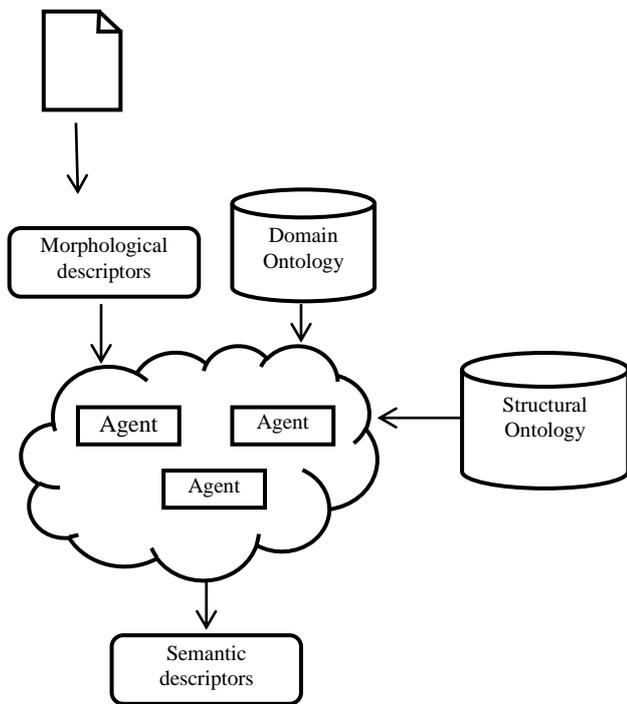


Fig. 2. Steps of solution

Agents have access to a domain ontology, structural ontology, morphological descriptors and electronic documents which will be indexed. Indexing process is produced on the sentences in the text. Sentences are processed sequentially by agents. The agents form a "team" to index the particular sentence. Thus, agents in the system after the start of the indexing are divided into teams.

C. Agent Types

The following types of agents are identified in the system, according to the functional separation:

Team Lead First Level Agent - TLFL agent,

Team Lead Second Level Agent - TLSL agent,

Word Indexer Agent - WI agent,

Index Writer Agent - IW agent.

The task of WI agent is accessing to the domain ontology and obtaining the set of possible semantic tags for the indexed word. An input word is passed to the WI agent for indexing with the parameters obtained at the stage of morphological analysis. Resulting set of possible semantic tags is passed to the TLSL agent.

TLSL agent binds to morphological descriptors of the sentence and distributes words to all available WI agents. TLSL agent finishes its work on the sentence when the consistent semantic descriptor is formed and written to the document. TLSL agent plans actions for the WI agents and participates in the auction for the resolution of contradictions. After building a consistent semantic descriptor TLSL agent

transmits the generated semantic descriptor of the sentence to IW agent who writes semantic tags to the document.

TLFL agent binds to morphological descriptors of the document and distributes descriptors of the sentences to all available TLSL agents. TLFL agent monitors the work of TLSL agents. If the work on the sentence is completed TLSL agent gives TLFL agent a new sentence. In addition, TLFL agent conducts an auction among TLSL agents to resolve ambiguity in the descriptors (see details in section «Agent negotiation»). Besides TLSL agents perform structural analysis. They distribute parts of structural ontology to TLSL agents.

D. Agent communication

Agents communicate through language FIPA ACL (Agent Communication Language developed by FIPA) [8]. Two types of actions are used. They are inform (inform about anything) and perform (execution of an action).

Inform action type is implemented in the following cases:

WI agent informs the TLSL agent of completion of indexing word and give it the set of possible semantic tags; content of the communication is as follows: (id, tags), where the id is the identifier word that came to be indexed, tags are returned set of possible semantic tags;

TLSL agent informs the TLFL agent of completion of indexing sentence with a specific identifier; content of this message contains an identifier of indexed sentence.

Perform action type is implemented in the following cases:

TLFL agent gives to the TLSL agent a task to index a sentence with a specific descriptor; content will look like this: (id, descriptor), where the id is the identifier of the sentence, descriptor is descriptor of the sentence received as a result of syntactic and semantic analysis;

TLSL agent gives a task to the WI agent to index a word with specific id; content will look like this: (id, word, parameters), where id is ID of the word, word is the word for indexing, parameters are parameters obtained at the stage of morphological and syntactic analysis;

TLSL agent gives a task to the IW agent to write semantic tag of specific word; content is as follows: (word, tag), where the word is an indexed word, tag is just a semantic tag of indexed word.

E. Planning

The planning is dynamic. TLSL agents themselves form a team of agents from the available WI agents. A count of needed WI agents depends on structure of a sentence. With a lack of WI agents at the time of formation of the team TLSL agent may designate to perform indexing of few words at once to the same WI agent. TLFL agent monitors the performance of work of TLSL agents and if they are released it assigns them new sentences for indexing. Completing of work of the agents (WI and TLSL) monitored not only by sending their corresponding messages of inform type, but also change their states (agent states) in the meaning of "vacant."

F. Agent knowledge bases

WI agents and IW agents are primitive reflex agents working in the mode of stimulus-response. Their main function is a simple, no inference, execution of work. In the knowledge bases of these agents are only procedural steps.

Knowledge bases of TLFL and TLSL agents represent productions with embedded procedural actions. In fact, the script actions are necessary for the distribution of work between agents. Accordingly TLSL agent knowledge base contains a script for word distribution among WI agents, and TLFL agent knowledge base includes a script for sentences distribution between agents TLSL.

G. Agent negotiation

TLFL agent conducts an auction among agents TLSL, each of which has a contextual memory (training component). Every TLSL agent using the contextual memory votes for a one option of semantic descriptor of the sentence. Option of semantic descriptor of the sentence with the highest number of votes will be considered as a true semantic descriptor of the sentence. The set of all consistent semantic descriptors of the sentences form the document semantic descriptor.

V. CONCLUSION

Unlike existing ontologies describing documents multidimensional ontology represents the document structure, which allows to consider this information during indexing process. In developing ontologies it included the mechanisms for integration with domain ontologies and expanding of ontology - adding new "aspects", which also expands the scope of the decision. The proposed multiagent approach creates preconditions for solving the optimization problem of parallel execution of semantic indexing.

Also planned that the developed ontology and algorithms will be used in a number of projects related to the development of domain-specific languages (Domain Specific Languages, DSL) for different domains based on linguistic tools MetaLanguage.

VI. ACKNOWLEDGEMENTS

The reported study was supported by RFBR, research project No. 14-07-31273.

REFERENCES

- [1] Segaran T., Evans C., Taylor J. Programming the Semantic Web, O'Reilly Media, 2009.
- [2] Lukashevich N.V., Dobrov B.V. Bilingual information retrieval based on the automatic conceptual indexing // Computational linguistics and intelligent technologies. Proceedings of the International Conference "Dialogue-2003". Protvino. June 11-16 2003y. / Ed. by I.M.Kobozevov, N.I.Laufer, V.P.Selegeya - M.: Science, 2003. - pp.425-432.
- [3] CNXML/DocumentOntology <http://mathweb.org/wiki/CNXML/DocumentOntology>
- [4] Dublin Core Metadata Element Set, Version 1.1 <http://dublincore.org/documents/dces/>
- [5] Document Ontology (draft) <http://www.cs.umd.edu/projects/plus/SHOE/onts/docmnt1.0.html>
- [6] Grishman. R. TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA, 1997. http://www.itl.nist.gov/div894/894.02/related_projects/tipster/.
- [7] Muninn Documents Ontology <http://rdf.muninn-project.org/ontologies/documents.html>
- [8] XML Languages <http://cnx.org/help/authoring/xml>
- [9] Lanin VV Lyadova LN Technology of support maintenance of electronic administrative regulations based on ontological models // Proceedings of the All-Russian conference with international participation "Knowledge-Ontology-Theory." Novosibirsk, 2011, pp. 38-46 V.2.
- [10] Lanin V.V. Using multi-level ontology repository for electronic document analysis // Proceedings of international scientific conference "Intelligent systems» (AIS'08) and "Intelligent CAD» (CAD-2008). Scientific publication in 4 vols. T. 1. - Moscow: Fizmatlit 2008. Pp. 202-206.
- [11] Program for morphological analysis of text in Russian "Mystem". [Electronic resource] [Mode of access:<http://company.yandex.ru/technologies/mystem/>] [Checked at: 24.06.12]

Generation of Domain-Specific Languages on the Basis of Ontologies

Alexander O. Sukhov

Department of Business Informatics
National Research University Higher School of Economics
Perm, Russian Federation
E-mail: Sukhov.psu@gmail.com

Abstract — Usage of visual domain-specific languages in software engineering allows to simplify the process of software creation and to attract to it the experts in domain, who are not professional programmers. However creation new domain-specific language is the nontrivial task, therefore the problem of automation of their development process is the topical task. For the automation, designing of visual modeling languages it is offered to use the ontologies received as a result of the analysis of text corpus. In article, the approach to automatic creation of visual modeling languages on the basis of domain ontologies is considered.

Keywords — *domain-specific modeling languages; ontologies; DSM-platform; MetaLanguage; metagraphs.*

I. INTRODUCTION

The domain-specific modeling languages (DSMLs), which are designed to solve a particular class of problems in a specific domain, are increasingly used at software development and maintenance process. Unlike the general-purpose languages, DSMLs are more expressive, easy to use and clear to various categories of users, as they operate with domain terms, which are familiar to users [1, 2]. For this reason now a large number of DSMLs is designed for creation of systems in different domains: artificial intelligence systems, distributed systems, mobile applications, real-time and embedded systems, simulation systems, etc. [3-7].

Despite of all DSMLs advantages they have one big disadvantage – complexity of their designing. If general-purpose languages allow to create models irrespectively to domain, in case of DSMLs for each domain, and in some cases, for each task, it is necessary to create a new domain-specific language. Another shortcoming of visual domain-specific language is that it is necessary to create convenient graphical editors to work with it. Therefore, a problem of automation of DSMLs development process is rather topical.

To support the process of development and maintenance of DSMLs the special kind of software – *language workbench (DSM-platform)* is used. Usage at DSMLs creation of a language workbench considerably simplifies the process of their designing. There are various DSM-platforms for creating visual DSMLs with the ability of determining user's graphical notation: MetaEdit+, Microsoft DSL Tools, Eclipse GMF,

QReal, etc. However, these tools do not allow to automatic create DSMLs.

This problem can be solved by the development of methods and tools, which will allow on the basis of a set of documents, available in domain, to build a conceptual domain model and automatically design a modeling language, which corresponds to singularities of domain, needs of various categories of users [8].

Let's consider the most advanced language workbenches [9].

II. RELATED WORKS

MetaEdit+ is a multiplatform language workbench that enables users to simultaneously work with several projects, each of which can have a few models [10]. At usage of this DSM-platform, besides a possibility of domain-specific language creation, the developer receives the CASE tool into which this language is integrated. MetaEdit+ allows to use several DSMLs at system creation.

The approach based on metamodels (models of modeling languages) interpretation, instead of code generation, which is used in MetaEdit+ allows changing the DSMLs definition at run-time.

DSL Tools [11] and Eclipse GMF [12] technologies provide the user with advanced IDE MS Visual Studio and Eclipse respectively. State Machine Designer [13], in fact, is an add-on to DSL Tools, which eliminates some of its shortcomings. However, the State Machine Designer allows creating DSMLs only using UML Activity Diagrams that considerably limits the range of solving tasks.

Multiplatform system QReal [14, 15] allows to define metamodels both in visual and textual view, therefore developers have a possibility to select the most suitable for them format of language description representation. Availability of an interpreter of behavioral diagrams and a debugger of the generated code puts this system in one row with tools MS DSL Tools, Eclipse GMF, which use for these purposes IDE.

This paper is supported by the Russian Foundation for Basic Research (grant 14-07-31330).

The analysis of DSM-platforms has revealed the following main restrictions inherent in the majority of the considered systems [16]:

1. Impossibility of multilevel modeling. Presence of such possibility would allow making changes at metalanguage description, to extend it with new constructions, thus bringing the metalanguage to the specifics of domain.
2. Modification of DSMLs description leads to necessity of regeneration of language editor: for modification DSMLs at first it is necessary to change its metamodel, to regenerate the source code of the editor, and only then it is possible to begin build models.
3. “Excess” functionality of the language workbench, which is not used at DSMLs creation. This functionality complicates the study of tools by the users, which are not professional programmers.
4. Lack of tools of horizontal models transformation. These means allow not only to create unified system description on the basis of the models constructed at various stages of system development, but also to generate source code according to user-specified template or to make conversion of the model described with one modeling language to model fulfilled in other graphical notation.

Moreover, the considered tools do not allow to fulfill automatic construction of DSMLs on the basis of domain analysis. This possibility allows to:

- simplify the process of DSMLs creating;
- create DSMLs, approached to specific domain;
- attract to DSMLs development process the users, who are not the professional programmers.

Thus, it is possible to say that creation of methods and tools of automatic DSMLs creation is the topical task, the solution of which will significantly simplify the process of visual domain-specific languages designing for various domains, and also will submit a possibility of involvement the experts in the process of development and maintenance of information systems.

For development of methods and tools for automatic of visual DSMLs designing, it is necessary to solve the following tasks:

1. Construct a mathematical model, which will allow to unify describe domain ontologies and metamodels of visual languages.
2. Develop rules of ontologies transformation in constructions of visual modeling languages.
3. Develop metrics and methods of comparing of DSMLs, which submit a possibility to estimate proximity of generated automatically domain-specific languages to domain specificity.
4. Implement the developed methods in dynamic library.

5. Integrate the created library into the MetaLanguage system.
6. Perform approbation of the received results by development of visual modeling languages for various domains.

III. METALANGUAGE SYSTEM

The MetaLanguage system eliminates some restrictions of the considered DSM-platforms.

This language workbench is designed to create visual dynamic adaptable domain-specific modeling languages, construct models using these languages and transform of the created models in various textual and graphical notations [17, 18].

A. Metalanguage of MetaLanguage System

One of the basic elements of language workbench is the *metalanguage* (meta-metamodel) – language for describing of other languages (metamodels). Thanks to presence of metalanguage, the DSM-platform allows to create domain-specific languages for the various domains that operate with familiar to user concepts. The main difference between metalanguages of MetaLanguage system from the MOF (Meta Object Facility) approach, used in the majority of DSM-platforms, is that thanks to interpretation of models at various abstraction levels, instead of the source code generation on their basis, it is possible to modify of DSML’s constructions in dynamic, during models creation. Besides, the process of metamodel creation becomes multilevel, so having defined a metamodel and having selected it as a metalanguage, the developer can use this meta-metamodel for creation of other metamodels, and this process can be infinite.

The basic elements of the metalanguage of MetaLanguage system are entity, relationship and constraint [19].

The *entity* describes a particular construction of modeling language, i.e. it is the domain object, important from the point of view of the solving problem.

Visual language constructions in rare cases exist independently, more often they are in some way related to each other, therefore at metamodel creation importantly not only to define the basic language constructions, but also correctly specify the relationships. The *relationship* is used for describing a physical or conceptual links between entities. Metamodel allows to create three types of relationships: *association*, *aggregation*, *inheritance*.

In practice quite often, there are cases when it is necessary to impose some *constraints* on entities and relationships between them. Some of constraints are set by metamodel structure, and others are described on some language.

Let’s consider an example. Fig. 1 shows a fragment of metamodel for UML Use Case diagrams. The metamodel contains two entities “Actor” and “Use Case”.

The entity “Use Case” has following attributes: “Name”, “Description”, “Creation_Date”. The attribute “Name” has a string type and defines the Use Case name. The attribute

“Description” sets the short description of the Use Case. The attribute of entity “Actor” is a string attribute “Name”, which specifies the name of the Actor.

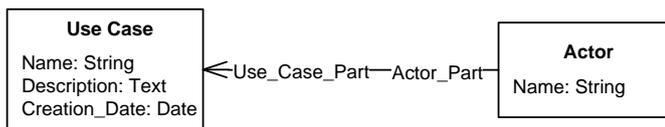


Fig. 1. Fragment of metamodel for UML Use Case diagrams

B. Architecture of MetaLanguage System

The architecture of MetaLanguage system is presented in Fig. 2. Uniform storage of all information about the system is the *repository*. It contains information about metamodels, models, entities, relationships, attributes, constraints. Information about the models and metamodels is stored uniformly, that allows to work with it by a single tool. The *browser of models* allows to load/save metamodels together with the models, created on their basis, to fulfill over metamodels and models various operations (editing, constraint checking, transformation, etc.). The *graphical editor* is the component, which provides the user the tools for metamodels and models creation. The *validator* allows to check constraints specified by user at metamodel describing. The *transformer* is the component that provides the ability to fulfill horizontal transformations of models to text on target programming language or to visual models, described in other graphical notation.

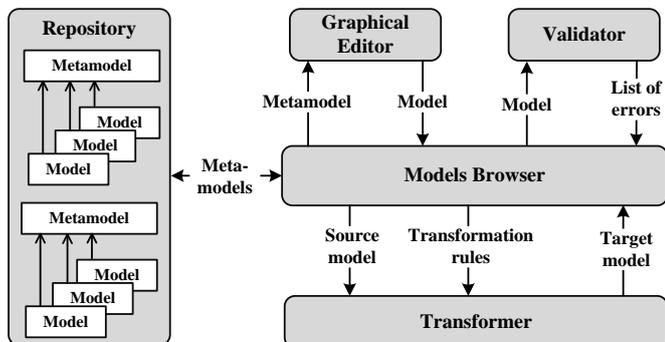


Fig. 2. Architecture of MetaLanguage system

Having described the basic components of a MetaLanguage system, let's consider how visual domain-specific modeling languages are designed.

Process of DSML definition begins with metamodel creation. For this purpose, it is necessary to specify the main constructions of created language, to define relationships

between them, to set constraints imposed on the metamodel entities and relationships. After building of metamodel the developer gets a customizable extensible visual modeling language.

Then using created DSML, the user should design models containing objects that describe specific domain concepts and links between them.

The validator should check up whether model satisfies to constraints, which were imposed on metamodel elements.

Then the developer can save the constructed metamodels and models in the form of XML-files or transform these models to other textual or graphical notation.

At metamodel modification, the system automatically makes all necessary changes in the models, which are created on the basis of this metamodel.

For automatic creation of DSMLs, it is necessary that the language workbench on the basis of domain description fulfilled creation of language's metamodel. Since the structure of the metalanguage of MetaLanguage system is similar to ontologies description languages, then it was decided to use ontologies as the basis of automatic DSMLs designing. There are various systems for the automatic construction of ontologies on the basis of a text corpus: OwlExporter [20], OntoGrid [21], etc. These systems allow to fulfill ontology creation on the basis of initial set of documents. The resulting ontology will be used by the MetaLanguage system for automatic creation of visual DSMLs (see Fig. 3).

IV. AUTOMATIC CREATION OF DSMLs

Formally ontology is the tuple $O = \{T, R, I\}$, where

- T is the finite nonempty set of domain concepts;
- R is the finite set of relationships between concepts;
- I is the finite set of interpretation of ontology concepts and relationships.

Thus, it is possible to say that the ontology is a directed labeled graph. The metamodel of visual modeling language is a graph also. For this reason the formalism that allows to describe domain ontology and metamodel of visual DSML in unified form is a labeled graph.

There are several types of graphs that can be used for representation of visual languages and ontologies: the classical graphs, digraphs, multigraphs, pseudographs, hi-graphs, hypergraphs, metagraphs and others.

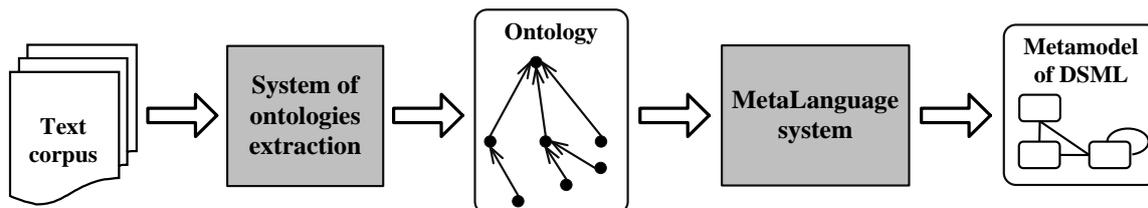


Fig. 3. Automatic creation of metamodel of DSML

As an analysis result of various types of graph it has been defined that the most appropriate formalism for describing the syntax of visual modeling languages in MetaLanguage system are pseudo-metagraphs [22].

Metagraph is an ordered pair $G = (V, E)$, where V is a finite nonempty set of nodes, E is a set of edges. Each edge $e_k = (V_i, V_j), V_i, V_j \subseteq V$ connects two subsets of nodes.

This type of graphs allows to reduce the number of graph arcs and to make the model more structured, logical. Therefore, in the metamodel graph the attributes and constraints of each entity and relationship are united in the separate sets. The model becomes more demonstrative, clear and corresponding to the solving task.

A. Metamodel Graph

Let's describe with usage of this formalism a metamodel of a visual modeling language.

Let $Ent = \{ent_i\}, i \in \aleph$ (\aleph is a set of natural numbers) is a set of metamodel entities, number of set elements is potentially unlimited, but at every fixed point in time is finite.

The set of metamodel relationships denotes as $Rel = \{rel_k\}, k \in \aleph$, number of set elements is potentially unlimited, but at every fixed point in time is finite.

Let's introduce the following designations:

- $EAttr_i = \{eattr_{ij}\}, i = 1, |Ent|, j \in \aleph$ is the set of metamodel graph nodes, which corresponds to entities attributes;
- $RAttr_k = \{rattr_{kl}\}, k = 1, |Rel|, l \in \aleph$ is the set of metamodel graph nodes, which corresponds to relationships attributes;
- $ERest_i = \{erest_{ij}\}, i = 1, |Ent|, j \in \aleph$ is the set of metamodel graph nodes, which corresponds to constraints imposed on entities;
- $RRest_k = \{rrest_{kl}\}, k = 1, |Rel|, l \in \aleph$ is the set of metamodel graph nodes, which corresponds to constraints imposed on relationships;
- $EEA = \{eea_i\}, i = 1, |Ent|$ is the set of metamodel graph arcs connecting each entity with the set of its attributes;
- $ERA = \{era_k\}, k = 1, |Rel|$ is the set of metamodel graph arcs connecting each relationship with the set of its attributes;
- $EER = \{eer_i\}, i = 1, |Ent|$ is the set of metamodel graph arcs connecting each entity with the set of its constraints;

- $ERR = \{err_k\}, k = 1, |Rel|$ is the set of metamodel graph arcs connecting each relationship with the set of its constraints;
- $EERR = \{eerr_i\}, i \in \aleph$ is the set of arcs corresponding to links between entities and relationships.

The number of elements of sets $EAttr_i, RAttr_k, ERest_i, RRest_k, EEA, ERA, EER, ERR, EERR$ potentially is not limited, but it is finite at every fixed point in time.

The metamodel graph is the directed pseudo-metagraph $GMM = (V, E)$, where V is a nonempty set of graph nodes, E is set of graph arcs and these sets are defined by (1) and (2):

$$V = Ent \cup \left(\bigcup_{i=1}^{|Ent|} EAttr_i \right) \cup \left(\bigcup_{i=1}^{|Ent|} ERest_i \right) \cup \left(\bigcup_{k=1}^{|Rel|} RAttr_k \right) \cup \left(\bigcup_{k=1}^{|Rel|} RRest_k \right), \quad (1)$$

$$E = EEA \cup EER \cup ERA \cup ERR \cup EERR. \quad (2)$$

Let's consider an example. We will construct a metamodel graph for the entity "Use Case" of UML Use Case diagrams. Metamodel of this diagram type is shown in Fig. 1. Attributes of the entity "Use Case" are "Name", "Description", "Creation_Date", i.e. for given entity

$$EAttr_i = \{\text{"Name"}, \text{"Description"}, \text{"Creation_Date"}\}.$$

The metamodel graph corresponding to a fragment of the "Use Case" entity is shown in Fig. 4.

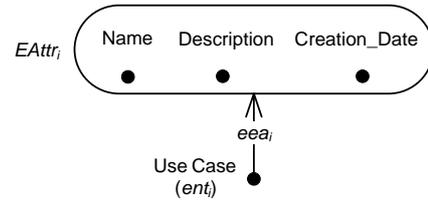


Fig. 4. Fragment of metamodel graph for "Use Case" entity

As can be seen from the figure

$$ERest_i = \emptyset, EEA = \{eea_i\}, EER = \emptyset, EERR = \emptyset.$$

B. Ontology Graph

Let's introduce the following designations:

- $T = \{t_i\}, i \in \aleph$ is the set of ontology graph nodes, which corresponds to concepts of ontology;
- $OAttr_i = \{oattr_{ij}\}, i = 1, |T|, j \in \aleph$ is the set of ontology graph nodes, which corresponds to concept attributes, which are not relationships;
- $OInst_i = \{oinst_{ij}\}, i = 1, |T|, j \in \aleph$ is the set of ontology graph nodes, which corresponds to concept instances;

- $ORel = \{orel_i\}, i \in \aleph$ is the set of ontology graph nodes corresponding to relationships between concepts.
- $OA = \{oa_i\}, i = 1, |T|$ is the set of ontology graph arcs connecting each concept with the set of its attributes;
- $OI = \{oi_i\}, i = 1, |T|$ is the set of ontology graph arcs connecting each concept with the set of its instances;
- $TR = \{tr_i\}, i \in \aleph$ is the set of ontology graph arcs corresponding to links between concepts and relationships.

The number of elements of sets $T, OAttr_i, OInst_i, ORel, OA, OI, TR$ potentially is not limited, but it is finite at every fixed point in time.

The *ontology graph* is the directed pseudo-metagraph $GO = (VO, EO)$, where VO is a nonempty set of graph nodes, EO is set of graph arcs and these sets are defined by (3) and (4):

$$VO = T \cup \left(\bigcup_{i=1}^{|T|} (OAttr_i \cup OInst_i) \right) \cup ORel, \quad (3)$$

$$EO = OA \cup OI \cup TR. \quad (4)$$

C. Mapping of Ontology Graph to Metamodel Graph

At mapping of a domain ontology to a modelling language metamodel it is necessary to fulfill the following algorithm:

1. To eliminate synonymy (to merge nodes containing synonymic concepts).
2. To delete instances of concepts and “is instance” relationships.
3. For each concept of ontology to create in a metamodel an entity with the concept’s attributes.
4. For each “is a” relationship of ontology to create an inheritance relationship in a metamodel.
5. For each “is part of” relationship of ontology to create an aggregate relationship in a metamodel.
6. For other relationships of ontology to create an association relationship in a metamodel.

According to this algorithm, let’s determine the mapping of the ontology graph to the metamodel graph, this mapping corresponds to operation of automatic creation of the metamodel graph.

Let’s introduce the following designations:

- $OInh = \{oinh_i\}, i \in \aleph$ is the set of ontology graph nodes corresponding to “is a” relationships between concepts;
- $OAggr = \{oaggr_i\}, i \in \aleph$ is the set of ontology graph nodes corresponding to “is part of” relationships between concepts;

- $OInstR = \{oinstr_i\}, i \in \aleph$ is the set of ontology graph nodes corresponding to “is instance” relationships between concepts;
- $RInh = \{rinh_i\}, i \in \aleph$ is the set of metamodel graph nodes corresponding to inheritance relationships;
- $RAggr = \{raggr_i\}, i \in \aleph$ is the set of metamodel graph nodes corresponding to aggregation relationships;
- $RAssoc = \{rassoc_i\}, i \in \aleph$ is the set of metamodel graph nodes corresponding to association relationships between concepts.

The number of elements of sets $OInh, OAggr, OInstR, RInh, RAggr, RAssoc$ potentially is not limited, but it is finite at every fixed point in time.

The mapping $conc: T \rightarrow Ent$ for each ontology concept puts in correspondence a metamodel entity.

The mapping $attr: OAttr \rightarrow EAttr$ for each ontology concept’s attribute puts in correspondence a metamodel entity’s attribute.

The mapping $inh: OInh \rightarrow RInh$ for each ontology “is a” relationship puts in correspondence a metamodel inheritance relationship.

The mapping $aggr: OAggr \rightarrow RAggr$ for each ontology “is part of” relationship puts in correspondence a metamodel aggregation relationship.

The mapping $assoc: ORel / (OInh \cup OAggr \cup OInstR) \rightarrow RAssoc$ for each other ontology relationship puts in correspondence a metamodel association relationship.

The mapping $ar: OA \rightarrow EEA$ for each arc oa_i of ontology graph puts in correspondence an arc eea_i of metamodel graph.

The mapping $rr: TR \rightarrow EERR$ for each arc tr_i of ontology graph puts in correspondence an arc err_i of metamodel graph.

Thus, the creation of the metamodel graph on the basis of ontology graph is determined by mappings $conc, attr, inh, aggr, assoc, ar, rr$.

D. “Smart House” Description Language

Let’s consider an example. Suppose that it is necessary to construct a visual modeling language for creation of models of “Smart House” systems.

At first, let’s analyze the components, which can be a part of “Smart House” systems. The basic elements of systems of this type are:

- life-support systems: heating, air conditioning and ventilation, lighting, security;
- sensors (devices that are responsible for obtaining of various readings and their sending to a central panel);

motion, leakings, fire and a smoke, closing/opening of object;

- system management tools: voice control, remote control (from a remote computer, phone, etc.), touch control (control by using of the touch screen of a central panel);
- central panel, which is responsible for receiving of data from sensors, management of life-support systems and obtaining of commands from the user.

The ontology received as a result of the analysis of a text corpus is presented in Fig. 5.

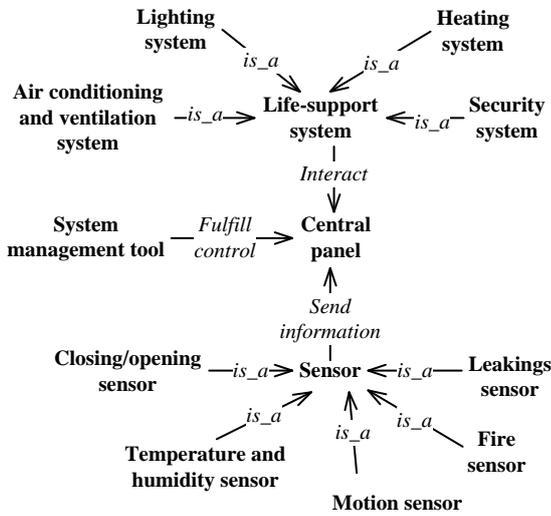


Fig. 5. Ontology of the structure of “Smart House” system

Concept “Life-support system” is the parent for concepts “Heating system”, “Air conditioning and ventilation system”, “Lighting system”, “Security system”. Concept “Sensor” is the parent for concepts corresponding to all types of system sensors. The relationship “Send information” connects the

concept “Sensor” with the concept “Central panel”. The relationship “Interact” describes the interaction of the concepts “Life-support system” and “Central panel”. The relationship “Fulfill control” connects the concepts “System management tool” and “Central panel”.

In this case, according to the considered earlier algorithm, the MetaLanguage system has constructed the metamodel presented in Fig. 6.

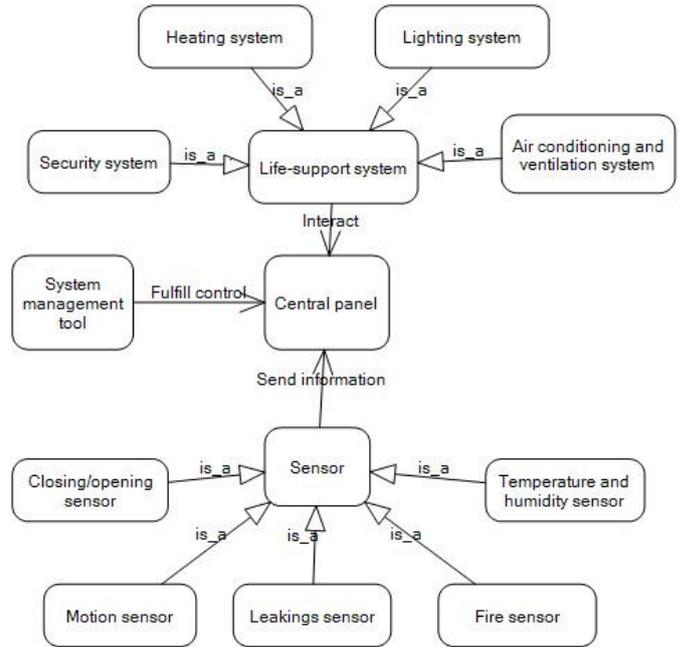


Fig. 6. Metamodel of “Smart House” Description Language

Fig. 7 shows one of many possible models of “Smart House” system, constructed in MetaLanguage system with the usage of designed DSML.

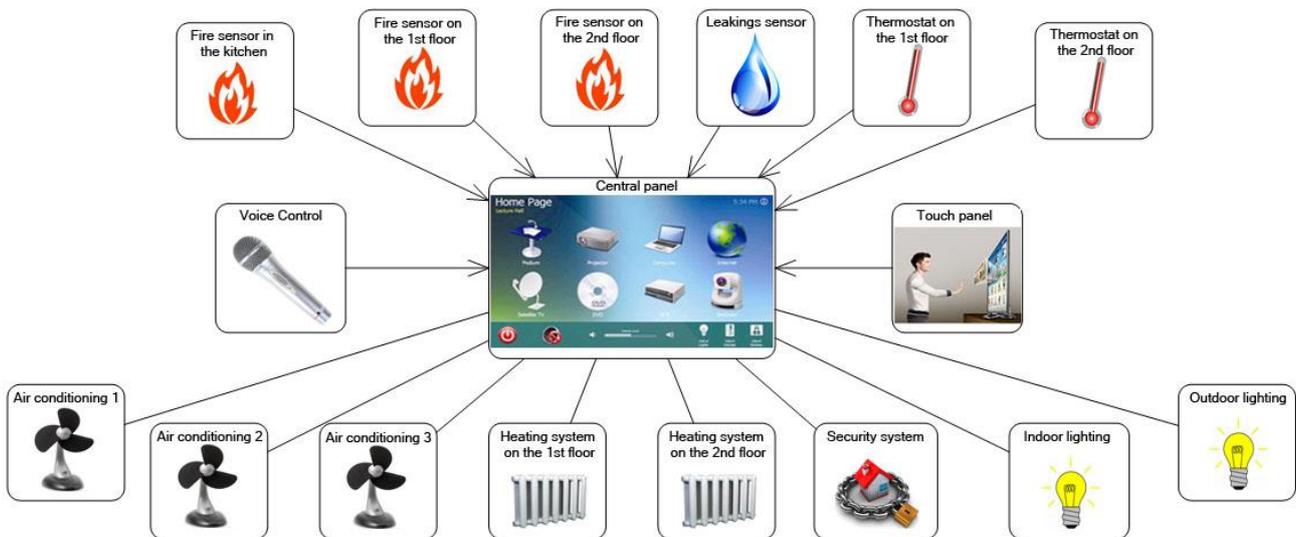


Fig. 7. Model of “Smart House” system

V. CONCLUSION AND FUTURE WORKS

The DSMLs creation is a difficult task. To automate the designing of visual modeling languages the approach, allowing on the basis of domain ontology to generate a language description, is offered. It allows reducing complexity and time of DSMLs development, and gives the chance to nonprofessional programmers to develop their own languages. In the future, it is planned to develop metrics and methods of comparing of DSMLs, which submit a possibility to estimate proximity of generated automatically domain-specific languages to domain specificity and to generate visual DSMLs for other domains.

REFERENCES

- [1] Hutchinson J., Rouncefield M., Whittle J. Model driven engineering practices in industry. Proceedings of the 33rd International Conference on Software Engineering, New York, 2011, pp. 633–642.
- [2] Velter M. MD*/DSL best practices Update March 2011. Available at: <http://www.voelter.de/data/pub/DSLBestPractices-2011Update.pdf>.
- [3] Bryksin T.A., Litvinov YU.V. Sreda vizual'nogo programmirovaniya robotov QReal: Robots. Materialy mezhdunarodnoj konferentsii "Informatsionnye tekhnologii v obrazovanii i nauke", 2011, pp. 332-334 (in Russian).
- [4] Erwig M., Walkingshaw E. A DSL for Explaining Probabilistic Reasoning. Proceedings of the 2nd international conference on Software Language Engineering, 2009, pp. 164-173.
- [5] Mezhuiev V.I. Predmetno-orientirovanoe modelirovanie raspredelennykh prilozhenij real'nogo vremeni. Sistemy obrabotki informatsii, 2010, no. 5(86), pp. 98-103 (in Russian).
- [6] Walter R., Masuch M. PULP Scription: A DSL for Mobile HTML5 Game Applications. Proceedings of the 11th International Conference on Entertainment Computing, 2012, pp. 504-510.
- [7] Sukhov A.O. Integratsiya sistem imitatsionnogo modelirovaniya i predmetno-orientirovannykh yazykov opisaniya biznes-protsessov. Matematika programmykh sistem, 2009, vol. 6, pp. 79-84 (in Russian).
- [8] Sukhov A.O. Razrabotka predmetno-orientirovannykh yazykov na osnove ontologij. Sbornik tezisov konferentsii "Sovremennye problemy matematiki i ee prikladnye aspekty", 2013, pp. 45-45 (in Russian).
- [9] Sukhov A.O. Sravnenie sistem razrabotki vizual'nykh predmetno-orientirovannykh yazykov. Matematika programmykh sistem, 2012, vol. 9, pp. 84-111 (in Russian).
- [10] Tolvanen J.-P., Pohjonen R., Kelly S. Advanced Tooling for Domain-Specific Modeling: MetaEdit+. Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA, 2007, pp. 48-55.
- [11] Cook S., Jones G., Kent S., Wills A.C. Domain-Specific Development with Visual Studio DSL Tools. Reading: Addison-Wesley, 2007, 560 p.
- [12] Kelly S. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications at OOPSLA, 2004, pp. 87-96.
- [13] Larionov A.V. Razrabotka vizual'nogo yazyka avtomatnogo programmirovaniya. Available at: <http://is.ifmo.ru/papers/StateMachineDesigner.pdf> (accessed 3 April 2014) (in Russian).
- [14] Terekhov A.N., Bryksin T.A., Litvinov YU.V. QReal: platforma vizual'nogo predmetno-orientirovannogo modelirovaniya. Programmnaya inzheneriya, 2013, no. 6, pp. 11-19 (in Russian).
- [15] Terekhov A.N., Bryksin T.A., Litvinov YU.V., Smirnov K.K., Nikandrov G.A., Ivanov V.YU., Takun E.I. Arkhitektura srede vizual'nogo modelirovaniya QReal. Sistemnoe programmirovanie, 2009, vol. 4, pp. 171-196 (in Russian).
- [16] Sukhov A.O. Sravnenie sistem razrabotki vizual'nykh predmetno-orientirovannykh yazykov. Matematika programmykh sistem, 2012, no 9, pp. 84-111 (in Russian).
- [17] Sukhov A.O., Lyadova L.N. MetaLanguage: a Tool for Creating Visual Domain-Specific Modeling Languages. Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering, 2012, pp. 42-53.
- [18] Sukhov A.O. Instrumental'nye sredstva sozdaniya vizual'nykh predmetno-orientirovannykh yazykov modelirovaniya. Fundamental'nye issledovaniya, 2013, no 4, vol. 4, pp. 848-852 (in Russian).
- [19] Lyadova L.N., Sukhov A.O. Yazykovej instrumentarij sistemy MetaLanguage. Matematika programmykh sistem, 2008, vol. 5, pp. 40-51 (in Russian).
- [20] Witte R., Khamis N., Rilling J. Flexible Ontology Population from Text: The OwlExporter. Available at: http://www.lrec-conf.org/proceedings/lrec2010/pdf/932_Paper.pdf.
- [21] Gusev V.D., Zavertajlov A.V., Zagorujko N.G., Kovalyov S.P., Nalyotov A.M., Salomatina N.V. System "OntoGrid" for construction of ontologies. Available at: <http://www.dialog-21.ru/Archive/2005/Zagoruiko%20Gusev%20Zavertailov/ZagoruykoNG.htm> (in Russian).
- [22] Sukhov A.O. Analiz formalizmov opisaniya vizual'nykh yazykov modelirovaniya. Sovremennye problemy nauki i obrazovaniya, 2012, no 2. Available at: <http://www.science-education.ru/102-5655> (in Russian).

Dynamic Information Model Interactions: design and implementation of database-driven workflow approach

Aleksej Petrov
Yaroslavl State University
Yaroslavl, Russia
e-mail: axel_petroff@mail.ru

Abstract—Workflow approach to create complex data processing mechanisms in object-oriented DBMS DIM is considered. Focusing on design and implementation, background and reasons of interactions research are discussed.

I. INTRODUCTION

A. DIM overview

Dynamic information model or DIM is an experimental database management system (DBMS), which investigation was caused by the desire of creation fully-operational object-oriented (OO) database.

Today, there are databases of two most popular types widely presented in the market and available for users:

- 1) relational;
- 2) object-oriented.

Relational databases in most cases has top performance between all others, but also has a great pitfall — insufficient flexibility. In contrast with them, OO databases can often be very flexible, but has serious drawbacks concerning data manipulations techniques and data integrity maintenance mechanisms.

In general, DIM developing is an attempt to combine best features from relational and objective databases. As in any OO system, in DIM, data is presented in the form of classified objects. Specialized query language ODQL was designed for DIM to give users ability to query and modify data [1]. Current implementation uses underlying relational database as a physical data storage [2].

Most important features of DIM are:

- 1) data model is described by finite amount of objects, which can have their own lifetimes;
- 2) every object is described by a set of properties;
- 3) every class of objects can be considered as a set of objects;
- 4) classes of objects can be linked with each other using some relations: *inheritance*, *inclusion*, *interaction* or *history*;
- 5) relations of classes can produce respective relations between objects of those classes or *inner inheritance* and *inner inclusion* relations between that objects;

- 6) object definition contains not only values of its properties, but also depends on links between objects and values of inherited properties and inherited links;
- 7) objects properties are changed in discrete points in time and determined by deterministic laws of objects interactions;
- 8) execution of interaction between objects (classes) can lead to massive dynamic changes of their objects (classes), which may end the lifetime of some objects (classes) and start the lifetime of other objects (classes).

Data querying (i.e. ODQL queries execution) algorithms and implementation mechanisms were finely described earlier in previous works. However, modern database systems are targeted not only to store & fetch data, they are also intended to use for processing data.

B. Problem statement

Data processing is simply the most important aspect of any computer program activity. In case of big amounts of data (which are usually stored in a database) it's desirable to put processing mechanisms as close to data as possible — inside the database.

We suppose that it is very important to have a user-friendly interface for developing database-driven application logic using standardized modules.

C. Database-oriented data processing approaches

If talking about relational database systems, it's widely known that not only SQL is used to manage data. Most RDBMS vendors exposed their systems to work with powerful programming languages, based on SQL. For example: PL\SQL from Oracle, T-SQL from Microsoft for SQL Server or PostgreSQL's PL/pgSQL. Those languages sometimes has a tricky syntax, but, together with triggers mechanisms, they give users a lot of possibilities to efficiently process data in different ways.

But, turning to object-oriented ones, it's not so easy to find out programming languages oriented on OO databases. For example, NoSQL databases, such as MongoDB, which mostly stores data, do not give users mechanisms to write data processing modules inside DB. Most well-known object-oriented databases, which use OQL [3] language as a primary data

management language provide limited mechanisms to combine queries with data manipulation instructions. For these systems it's possible to use embedded OQL code blocks (in O2 ODMG Database System) or embed objective queries into application code, written in some general purpose programming language: C++, Java, etc. But they also do not provide programming languages deeply integrated into database core.

As discussed above, widely used relational DBMSes usually provide ability to create application's logic in form of code modules, some OO DBMSes usually also do it in a limited way. Currently, DIM is following SQL approach: PL \ODQL and DIM-FL domain specific languages are injected to use them for processing data. Although, it is possible to inject application logic into database, in fact, in many cases this approach is not convenient.

D. Solution approach

Using modern declarative programming techniques, in simple terms, problem can be solved as adaptation of workflow application's design methodology for DIM database. Considered examples on databases currently do not provide such mechanisms out-of-box.

After analyzing these disadvantages, some big steps to integrate user-friendly & handy data processing mechanisms into DIM were taken, in details they are discussed further in the paper. Chapter 2 is devoted to describe existing DIM data processing techniques. In the 3rd chapter theoretical explanations of developing workflow approach are given, and finally, in the chapter 4, DIM infrastructure, workflow engine and IDE "DIM Developer", which has been created to develop application's logic within DIM, implementations are considered in detail.

II. DATA PROCESSING IN DIM: PL \ODQL, DIM-FL AND INTERACTIONS CONCEPTS

Although ODQL is expressive, relatively simple and powerful language, nevertheless it was developed only as data-management language. To perform more complex tasks of data processing 2 extra languages were introduced: PL \ODQL (Programming language for DIM) and DIM-FL (DIM Formula language).

A. PL \ODQL

PL \ODQL was developed as an extension of ODQL and has incorporated the best features from PL \SQL, Tutorial D and OQL languages [5]. Under the strong influence of PL \SQL, PL extension for ODQL was developed as procedural, to simplify studying for future developers. PL \ODQL gives developer a power of a full-valued programming language.

B. DIM-FL

DIM-FL is a specialized formula-style language, which was added to create even more simple mechanism than PL \ODQL to automate calculation intensive, but not complex tasks [6]. DIM-FL statements are treated as formulas, which are evaluated during formula module execution.

Such a division of responsibility helps to implement data processing algorithms in a more natural way: if a special

calculation standing order exists as a math formula, it is better to use DIM-FL, otherwise, if only a step-by-step algorithm exists — the best choice is PL \ODQL.

C. DIM interactions concepts

As it was mentioned above, DIM DBMS is objective; however, simplified procedural design of PL \ODQL and DIM-FL languages breaks down object-oriented methodology. To resolve this digression, were introduced DIM interactions.

Interaction is an atomic part of processing data within the DIM ecosystem. In a daily life's terms data processing is a *deal*. When we think about a *deal*, we always keep in mind that someone initiates a *deal*, processes a *deal* (usually a human), also *deal* is somehow processed by some processor (usually a mechanism), and a result of a *deal* is saved in some place. Thus, *deal processing* consists of four aspects. In terms of OO design principles, it is possible to encapsulate *deal processing* logic and all the objects, which take part in action, in a special class – interaction class.

Of course, in some ways it brakes native encapsulation, because such an approach splits classes data from theirs logic. Nevertheless, using Interactions makes it possible to return to pure object oriented design methodology, because user operates with objects only, there is no need to write complex procedural code on the top level of application programming infrastructure. Moreover, Interaction execution is symmetrical with respect to all objects it consist from.

Interactions are the key point of data processing in DIM.

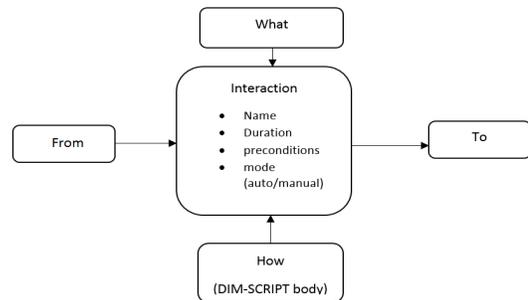


Fig. 1. DIM interaction relation scheme

To be clear, the object, which takes part in interaction can act in one of four roles:

- From
- To
- What
- How

From, *to* and *what* objects are presented in the form of ODQL queries. Unlike them, *how* class object is treated as a body of interaction and it is written in a special DIM-SCRIPT language.

DIM-SCRIPT is a simple language with several kinds of instructions, which drives Interaction execution. The language has 2 types of instructions:

- 1) Module launch instructions: **EXECUTE / CALCULATE** – to call PL \ODQL and DIM-FL modules within Interaction;
- 2) **WHAT / FROM / TO** instructions which point, with the help of ODQL queries, what objects take part in Interaction execution.

If objects, involved in Interaction are not set within Interaction body (obtained from **WHAT, FROM, TO** queries), they have to be passed as parameters from the outside.

Interactions are widely used by DIM framework itself. Manipulations with classes, objects, properties and many other actions are Interactions. This technique opens perspective to standardize auditing and logging mechanisms, and to simplify development of modifying actions.

III. MOVING TO WORKFLOW DESIGN

Although Interaction is a powerful abstraction over real life process, many processes cannot be presented as atomic operations. They often consist of many sequential or parallel steps.

The most understandable way to extend interactions functionality and flexibility is to aggregate them into oriented graphs. In terms of programming, application's logic graphs are often called workflows.

Graphs, by definition, consist of nodes and edges. In this case, interactions are treated as nodes. In addition, by design, such graphs have only one starting node (node without incoming edges). So, when first workflow node is called, it launches a chain of interactions executions.

Nodes are executed in order they are linked with each other using graph edges.

This approach also makes it possible to implement parallel tasks execution. However, this improvement has serious limits. For example: it is mostly impossible to correctly process same data object simultaneously by several interactions. Lets figure out main limitation for parallel executions of Interactions: objects, which act as *What* objects have to be different for every parallel task.

Graph are expected to include a set of parameters, which are used to store intermediate values, passed between interactions. These parameters should have particular type, or class, which can be defined by using **SELECT CLASS ... ODQL** query.

Now, it is time to introduce mechanism to control Interactions execution. As for single Interaction, it does not make serious sense when the Interaction was executed, because it does not trigger many changes. However, with workflows, this problem is critical: accidental launch can completely change data without possibility to stop the flow. To eliminate this, 2 modes of execution are introduced: *automatic* and *manual*. If Interaction is in *manual* execution mode, while executing workflow, executor process should freeze at this point, and wait until user approves this execution from database client. Interactions in this case are called *required*.

Such approach gives some pros:

- no need to use third-party software, implementing workflow engine;
- as close to data as possible, no need to transfer big amounts of data out of storage during data processing;
- great simplification in development comparing to developing and debugging very tricky SQL triggers chains.

IV. INTERACTIONS IMPLEMENTATION AND DESIGN

A. DIM infrastructure implementation

Interactions execute in a common DIM DBMS environment, which is in turn, a Java-based add-on to Oracle RDBMS running within a database server. DIM infrastructure consist of several modules:

- 1) Metalevel, provides translation between DIM objects and Oracle relational database records, it acts as a physical data storage and can be replaced with some other underlying data storage system;
- 2) ODQL queries translator and executor, provides data fetching and manipulating using textual queries [4];
- 3) PL \ODQL interpreter;
- 4) DIM-FL interpreter;
- 5) DIM-SCRIPT interpreter;
- 6) Interaction processor.

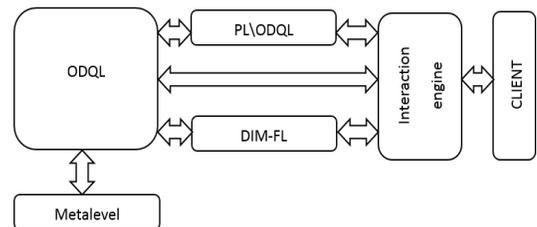


Fig. 2. DIM interactions infrastructure

Query translator (discussed in detail in [4]), is used to create inner objective form of queries. From textual query it produces query objects that should be cached for further usage.

Interpreters are currently pure interpreters, that executes statement-after-statement, but they are under intensive process of rewriting into native PL \ODQL / DIM-FL / DIM-SCRIPT to Java bytecode builders to improve performance.

B. Interactions implementation within DIM DBMS

In terms of DIM, interactions and interaction graphs are just special classes, which are stored in Metalevel, as any other plain DIM classes. Interactions are combined into graphs by using *inclusion* relation links which mark that interaction is a part of some graph and by using special *Link* class objects (included into graph class object) which reflect structure of graphs in order they link interactions with each other.

Moreover, interaction inside a graph can use objects not only from *from*, *to* or *what* queries result, but also parameters can be passed from graph execution process scope. To keep ability to use interactions in different graphs, information about parameter mapping is excluded from interactions objects.

Parameters descriptions are stored in *GraphParameter* objects, and mapping information is kept in *Mapping* objects, that are included in graph and linked with interactions which are parts of the graph. *Mapping* objects link graph parameters with interactions. *GraphParameter* objects link parameters inside interaction with graph parameters. Inner property means a role in what object should be mapped into interaction: *from*, *to* or *what*.

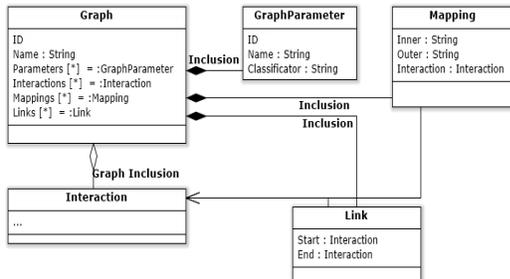


Fig. 3. Interactions graph classes scheme

The processor itself takes control over Interaction workflows execution state and over steps between nodes. In addition, it prevent conflicts with parallel execution by terminating incorrect graph execution process.

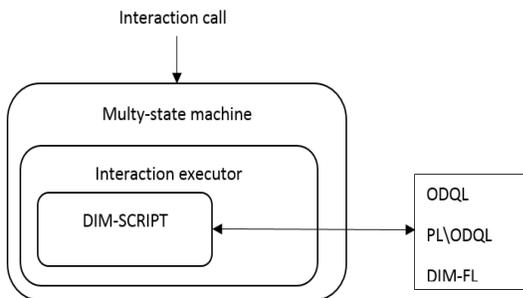


Fig. 4. Interactions processor

When processor receives a call to launch an interaction, first, it loads interaction class and then determines whether current interaction is a starting node of interaction graph or not. If its true, then the processor loads interaction graph (loads all containing interactions and builds structure from containing links objects). On the next step, control of execution moves to so-called state machine executor (because this machine can has multiple active states: interactions can be launched in parallel).

Execution of single interaction is delegated to Interaction executor. Single interaction execution process itself drops into several phases:

- 1) Determining objects, which are taking part in execution (objects in roles *what*, *to* and *from*), by launching specific ODQL queries or mapping outer parameters;
- 2) Checking whether objects in role *what* of interactions, which are executing in parallel, are not equal — to prevent data corruption;
- 3) Checking fulfillment of preconditions, by launching special *check* PL \ODQL procedure (if it breaks with exception — preconditions are not satisfied);

- 4) Execution of module, by delegating calls to PL \ODQL and DIM-FL interpreters.

When single interaction execution involves execution of interaction's graph, situation becomes a lot of trickier. Every user of database can call many interaction, and for each of them DIM instantiates state machine executor, that holds status of actively executing interactions and controls jumps between interactions of a graph. After finishing execution of the first node it makes a jump to consequent interaction nodes of a graph, and then single interaction executor repeats its job: check conditions and launch interaction's body. It's very important that interactions in graph can be marked as *manual*, i.e. they can be executed only manually by user. In this case, state machine executor pauses graph execution and stores its state in special *GraphState* class object. User can resume execution: he has to approve blocker interaction launch from the DIM database client to continue interaction's graph execution or reject them to kill paused interaction's graph execution. If execution was killed, then *GraphState* object is removed from DIM.

All actions, taken by interaction processor, are passed through auditing mechanisms who attentively log execution parameters into special *ExecutionLog* records: durations, times, objects, results and initiators.

C. Interactions design – DIM developer

Designing application logic within DIM DBMS is often a very elaborate task for user. Here are some difficulties:

- 1) writing complex queries to obtain necessary objects;
- 2) creating modules in different programming languages, such as PL \ODQL, DIM-FL and DIM-SCRIPT;
- 3) designing interaction workflow graphs in textual form, i.e. user has to manually create classes and links using ODQL queries.

To give users an opportunity to create DIM DBMS applications rapidly and easily, it became very important to develop convenient database-oriented IDE — "DIM Developer". "DIM Developer" is a standalone client application, written in Java and using Swing interface library, which connects to the DIM database server through the network protocols. It can be used as a database client to simply execute ODQL queries and display results, fetched from storage, but mostly it's intended to act as a rich developing environment.

This IDE allows user to manage DIM objects using friendly step-by-step interface. Using existing wizards, one can visually create classes, objects (DIM Navigator [7]), application modules and interactions. For example, interaction creation splits into three steps:

- 1) Choosing objects, which will take part in interaction execution. This operation can be accomplished with help of DIM Navigator or built-in ODQL query builder, or by using plain text editor to manually write the select queries.
- 2) Customization of interaction. The main purpose of this step is to define preconditions, fulfillment of

which will act as a gate at interaction launch. Moreover, it is very important to select execution mode: *automatic* or *manual*, because if *manual* execution mode is selected, then this interaction will become blocker node in workflows. Such blocker nodes are become *required* interactions — user need to approve these nodes' executions manually.

- 3) Developing the body of interaction. Using code editor developer should write some lines of code in DIM-SCRIPT, describing calls of PL \ODQL functions and DIM-FL modules.

Developing PL \ODQL modules in DIM Developer is quite similar to creating, for example, PL \SQL packages in well-known IDE Oracle PL \SQL developer. Programming DIM-FL modules is even simpler, due to possibility to use native visual math notation of formulas.

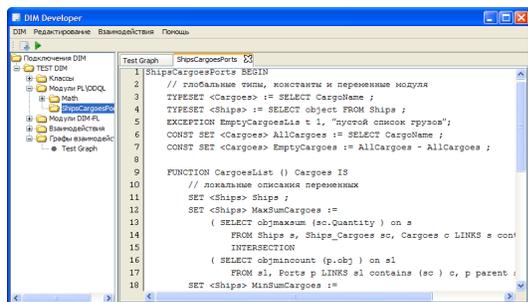


Fig. 5. Developing PL \ODQL module in DIM Developer

One of the most important feature of the IDE is the ability to visually create interaction graphs or workflows. Developer just picks up necessary interactions and links them using edges, and after these manipulations, one receives completely-operational business workflow.

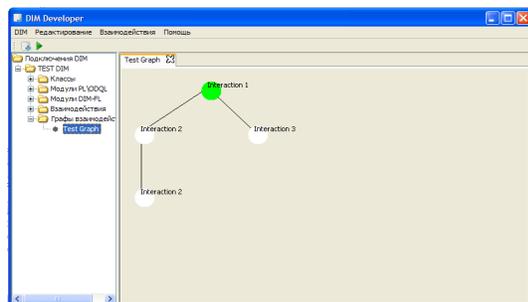


Fig. 6. Simple interactions graph developing example

V. CONCLUSION

As a result of this research was created object-oriented database framework, which implements database-driven workflow engine as a part of DBMS. Also were strictly postulated conditions for Interactions execution and steps between Interaction nodes in a special workflow interactions' graphs. For users convenience, was created a RAD tool "DIM Developer", which is currently a unique tool for DIM system that allows to intuitively create sophisticated application logic from ready-to-use Interactions or create new ones, using its rich infrastructure.

Considered approach of data processing organization can be very useful in case of dividing large amount of work between different specialists. Manager can give business logic creation work out to analysts (because in this case it isn't necessary to use programming languages), sophisticated mathematical calculations programming — to engineers, and developing of system logic – to developers. Moreover, all these tasks can be done within one system.

REFERENCES

- [1] Rublev V.S., Yazyk obektnykh zaprosov dinamicheskoy informatsionnoy modeli DIM // Modelirovanie i analiz informatsionnykh system.— Yaroslavl:YarGU, 2010, vol. 17, 3, pp. 144-161
- [2] Rublev V.S., Kajbyshev A.Sh. Organizatsiya khraneniya dannykh I vypolneniya zaprosov v dinamicheskoy informatsionnoy modeli DIM // Yaroslavskij pedagogicheskij vestnik.— 2012, vol. 3 (Estestvenne nauki), 1, pp. 7-20.
- [3] Hector Garcia-Molina. Database systems. The complete book // Pearson Prentice Hall.— 2008
- [4] Petrov A.N. ODQL query execution mechanism // Science Drive-2013: doklady molodezhnoj nauchnoj shkoly.— Yaroslavl:YarGU, 2013, pp. 99-103.
- [5] Rublev V.S., Petrov A.N. Yazyk PL \ODQL I mnozhestva s indeksami // Yaroslavskij pedagogicheskij vestnik.— 2012, vol 3 (Estestvenne nauki), 4, pp. 74-83.
- [6] Pisarenko D.S. Yazyk matematicheskikh formul DIM-FL Dinamicheskoy informatsionnoy modeli DIM // Stuchcheskie zametki po informatike i matematike: Sbornik nauchnykh statej studentov i aspirantov fakulteta IVT. Yaroslavl: YarGU, 2008, vol. 5, pp. 88-96.
- [7] Antonov D.V. Navigator DIM // Zametki po informatike i matematike: sbornik nauchnykh statej.— Yaroslavl: YarGU, 2013, 5, pp. 10-13.