

SYRCoSE 2013

Editors:

Alexander Kamkin, Alexander Petrenko,
Andrey Terekhov

Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on
Software Engineering

Kazan, May 30-31, 2013

2013

Proceedings of the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2013), May 30-31, 2013 – Kazan, Russia:

The issue contains the papers presented at the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2013) held in Kazan, Russia on 30th and 31st of May, 2013. 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 modeling of computer systems, software testing and verification, parallel and distributed systems, information search and data mining, image and speech processing and others.

Труды 7-ого весеннего/летнего коллоквиума молодых исследователей в области программной инженерии (SYRCoSE 2013), 30-31 мая 2013 г. – Казань, Россия:

Сборник содержит статьи, представленные на 7-ом весеннем/летнем коллоквиуме молодых исследователей в области программной инженерии (SYRCoSE 2013), проводимом в Казани 30 и 31 мая 2013 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: моделирование компьютерных систем, тестирование и верификация программ, параллельные и распределенные системы, информационный поиск и анализ данных, обработка изображений и речи и др.

ISBN 978-5-91474-020-4

Contents

Foreword.....	6
Committees / Referees.....	7
Formal Models of Computer Systems	
NPNtool: Modelling and Analysis Toolset for Nested Petri Nets <i>L. Dworzanski, D. Frumin</i>	9
The Tool for Modeling of Wireless Sensor Networks with Nested Petri Nets <i>N. Buchina, L. Dworzanski</i>	15
Process Mining and Trace Analysis	
DPMine: Modeling and Process Mining Tool <i>S. Shershakov</i>	19
Recognition and Explanation of Incorrect Behavior in Simulation-Based Hardware Verification <i>M. Chupilko, A. Protsenko</i>	25
Model Transformations	
Horizontal Transformations of Visual Models in MetaLanguage System <i>A. Sukhov, L. Lyadova</i>	31
An Approach to Graph Matching in the Component of Model Transformations <i>A. Seriy, L. Lyadova</i>	41
Testing Software and Hardware Systems	
Technology Aspects of State Explosion Problem Resolving for Industrial Software Design <i>P. Drobintsev, V. Kotlyarov, I. Nikiforov</i>	46
MicroTESK: An Extendable Framework for Test Program Generation <i>A. Kamkin, T. Sergeeva, A. Tatarnikov, A. Utekhin</i>	51
Probabilistic Networks as a Means of Testing Web-Based Applications <i>A. Bykau</i>	58
Software Mutation Testing: Towards Combining Program and Model Based Techniques <i>M. Forostyanova, N. Kushik</i>	62
Experimental Comparison of the Quality of TFSM-Based Test Suites for the UML Diagrams <i>R. Galimullin</i>	68
Linux Development and Verification	
Experience of Building and Deployment Debian on Elbrus Architecture <i>A. Kuyan, S. Gusev, A. Kozlov, Z. Kaimuldenov, E. Kravtsunov</i>	73
Generating Environment Model for Linux Device Drivers <i>I. Zakharov, V. Mutilin, E. Novikov, A. Khoroshilov</i>	77

On the Implementation of Data-Breakpoints Based Race Detection for Linux Kernel Modules <i>N. Komarov</i>	84
Software Engineering Education	
Mobile Learning Systems in Software Engineering Education <i>L. Andreicheva, R. Latypov</i>	89
Computer Networks	
Hide and Seek: Worms Digging at the Internet Backbones and Edges <i>S. Gaivoronski, D. Gamayunov</i>	94
Station Disassociation Problem in Hosted Network <i>A. Shal</i>	108
On Bringing Software Engineering to Computer Networks with Software Defined Networking <i>A. Shalimov, R. Smeliansky</i>	111
Parallel and Distributed Systems	
The Formal Statement of the Load-Balancing Problem for a Multi-Tenant Database Cluster With a Constant Flow of Queries <i>E. Boytsov, V. Sokolov</i>	117
Scheduling Signal Processing Tasks for Antenna Arrays with Simulated Annealing <i>D. Zorin</i>	122
Automated Deployment of Virtualization-Based Research Models of Distributed Computer Systems <i>A. Zenzinov</i>	128
Information Search and Data Mining	
Intelligent Search Based on Ontological Resources and Graph Models <i>A. Chugunov, V. Lanin</i>	133
Intelligent Service for Aggregation of Real Estate Market Offers <i>V. Lanin, R. Nesterov, T. Osotova</i>	136
An Approach to the Selection of DSL Based on Corpus of Domain-Specific Documents <i>E. Elokho, E. Uzunova, M. Valeev, A. Yugov, V. Lanin</i>	139
Computer Graphics and Image/Speech Processing	
Beholder Framework: A Unified Real-Time Graphics API <i>D. Rodin</i>	144
Image Key Points Detection and Matching <i>M. Medvedev, M. Shleyovich</i>	149
Voice Control of Robots and Mobile Machinery <i>R. Shokhirev</i>	155

Application-Specific Methods and Tools

Service-Oriented Control System for a Differential Wheeled Robot <i>A. Mangin, L. Amiraslanova, L. Lagunov, Yu. Okulovsky</i>	159
Scheduling the Delivery of Orders by a Freight Train <i>A. Lazarev, E. Musatova, N. Khusnullin</i>	165
Optimization of Electronics Component Placement Design on PCB Using Genetic Algorithm <i>L. Zinnatova, I. Suzdalcev</i>	169

Foreword

Dear participants, we are glad to meet you at the 7th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). The event is held in Kazan, the capital and largest city of the Republic of Tatarstan, Russia. The colloquium is hosted by Kazan National Research Technical University named after A.N. Tupolev (KNRTU), former Kazan Aviation Institute (KAI), one of the leading Russian institutions in aircraft engineering, engine- and instrument- design and manufacturing, computer science and radio- and telecommunications engineering. SYRCoSE 2013 is organized by Institute for System Programming of the Russian Academy of Sciences (ISPRAS) and Saint-Petersburg State University (SPbSU) jointly with KNRTU.

In this year, Program Committee (consisting of more than 40 members from more than 20 organizations) has selected 30 papers. Each submitted paper has been reviewed independently by three referees. Participants of SYRCoSE 2013 represent well-known universities, research institutes and companies such as Belarusian State University of Informatics and Radioelectronics, ISPRAS, Kazan Federal University, KNRTU, Moscow State University, National Research University Higher School of Economics, Perm State National Research University, Tomsk State University, Ural Federal University, V.A. Trapeznikov Institute of Control Sciences of the Russian Academy of Sciences, Yaroslavl State University and ZAO "MCST" (2 countries, 8 cities and 12 organizations).

We would like to thank all of the participants of SYRCoSE 2013 and their advisors for interesting papers. We are also very grateful to the PC members and the external reviewers for their hard work on reviewing the papers and selecting the program. Our thanks go to the invited speakers, Mirko Conrad (The MathWorks GmbH, Germany), Yuri Gubanov ("Belkasoft" and SPbSU, Russia) and Marek Miłosz (Institute of Computer Science, Lublin University of Technology, Poland). We would also like to thank our sponsors and supporters, Russian Foundation for Basic Research (grant 13-07-06008-Г), Cabinet of Ministers of the Republic of Tatarstan, Intel, Nizhny Novgorod Foundation for Education and Research Assistance and ICL-KME CS. Finally, our special thanks to local organizers Liliya Emaletdinova (Institute for Technical Cybernetics and Informatics, KNRTU), Kirill Shershukov (Academy for Information Technologies, KNRTU), Igor Anikin, Dmitry Kolesov, Mikhail Shleymovich and Dmitry Strunkin (KNRTU) for their invaluable help in organizing the colloquium in Kazan.

Sincerely yours

Alexander Kamkin, Alexander Petrenko, Andrey Terekhov
May 2013

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
Higher School of Economics

 Eduard Babkin – Russia
National Research University Higher School of Economics

 Svetlana Chuprina – Russia
Perm State National Research 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

 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

 Daniel Kurushin – Russia
State National Research Polytechnic University of Perm

 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

 Tiziana Margaria – Germany
University of Potsdam

 Igor Mashechkin – Russia
Moscow State University

 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

 Elena Pavlova – Russia
Microsoft Research

 Yuri Okulovsky – Russia
Ural Federal University

 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

 Ruslan Smelyansky – Russia
Moscow State University

 Nikolay Shilov – Russia
Ershov Institute of Informatics Systems

 Valeriy Sokolov – Russia
Yaroslavl Demidov State University

 Petr Sosnin – Russia
Ulyanovsk State Technical University

 Vladimir Voevodin – Russia
Research Computing Center of Moscow State University

 Mikhail Volkanov – Russia
Moscow State University

 Mikhail Volkov – Russia
Ural Federal University

 Nadezhda Yarushkina – Russia
Ulyanovsk State Technical University

 Rostislav Yavorsky – Russia
Skolkovo

 Nina Yevtushenko – Russia
Tomsk State University

 Vladimir Zakharov – Russia
Moscow State University

Organizing Committee Chairs and Secretaries

 Alexander Petrenko – Russia
Institute for System Programming of RAS

 Alexander Kamkin – Russia
Institute for System Programming of RAS

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

 Kirill Shershukov – Russia
Academy for Information Technologies, KNRTU

Referees

Eduard Babkin

Svetlana Chuprina

Liliya Emaletdinova

Victor Gergel

Efim Grinkrug

Maxim Gromov

Vladimir Hahanov

Shihong Huang

Alexander Kamkin

Vsevolod Kotlyarov

Oleg Kozyrev

Daniel Kurushin

Rustam Latypov

Alexander Letichevsky

Alexander Lipanov

Irina Lomazova

Ludmila Lyadova

Tiziana Margaria

Marek Miłosz

Valery Nepomniaschy

Mykola Nikitchenko

Yuri Okulovsky

Elena Pavlova

Alexander Petrenko

Ivan Piletski

Vladimir Popov

Yury Rogozov

Rustam Sabitov

Nikolay Shilov

Sergey Smolov

Valeriy Sokolov

Petr Sosnin

Andrey Tatarnikov

Andrey Terekhov

Dmitry Volkanov

Mikhail Volkov

Nadezhda Yarushkina

Rostislav Yavorskiy

Nina Yevtushenko

Vladimir Zakharov

NPNtool: Modelling and Analysis Toolset for Nested Petri Nets

Leonid Dworzanski

Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
leo@mathtech.ru

Daniil Frumin

Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
difrumin@edu.hse.ru

Abstract—Nested Petri nets is an extension of Petri net formalism with net tokens for modelling multi-agent distributed systems with complex structure. While having a number of interesting properties, NP-nets have been lacking tool support. In this paper we present the NPNtool toolset for NP-nets which can be used to edit NP-nets models and check liveness in a compositional way. An algorithm to check m-bisimilarity needed for compositional checking of liveness has been developed. Experimental results of the toolset usage for modelling and checking liveness of classical dining philosophers problem are provided.

Index Terms—Petri nets, nested Petri nets, multi-agent systems, compositionality, liveness

I. INTRODUCTION

In our world distributed, multi-agent and concurrent systems are used everyday to the point that we don't even notice them working for us. Not only civilian and military air and water carriers are equipped with hi-tech electronics and software, but even laundry machines, microwave ovens, refrigerators, air-condition systems and other implements are controlled by distributed software.

In the great amount of research on defining parallel and concurrent systems, in recent years a range of formalisms have been introduced, modified or extended to cover agent systems. One of such approaches, which gained widespread usage, is Petri nets. One downside of the classical Petri nets formalism is its flat structure, while multi-agent systems commonly have complex nested apparatus. This prevents us from easily specifying models of multi-agent systems in a natural way. The solution to this problem was found by R. Valk [12], who originated the net-within-nets paradigm. According to the nets-within-net paradigm [11], the tokens in a Petri net can be nets themselves. Usually, there is some sort of hierarchy among the networks: there is a *system net*, the top level network, and all other nets are assigned each to their *initial place*, providing us with the hierarchy of the nets in one big *higher-order net*.

One of the non-flat Petri net model is Nested Petri nets [9], [10], [7]. In nested Petri nets (NP-nets), there is a *system net*, in some places of which *element nets* resign, in the form of *net tokens*. NP-nets have internal means of *synchronization* between element nets and the system net.

The research is partially supported by the Russian Fund for Basic Research (project 11-01-00737-a).

But the application and evolution of the formalism is hampered by the lack of tool support. So far, there are no instruments (simulators, model checker software) which provide any kind of support for the nested Petri nets formalism. In this paper we present our newly developed project *NPNtool*.

The paper is organized as follows. To start with, we give some necessary foundations of Petri nets and nested Petri nets. After that we describe our toolset (both frontend and backend). We describe a simple experiment we've conducted and conclude the paper with the directions of future research.

II. PETRI NETS

In literature, there is a variety of definitions for Petri nets, a common one would be the following.

Definition 1. A Petri net (P/T-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;
- $W : F \rightarrow \mathbb{N} \setminus 0$ – an arc multiplicity function, that is, a function which assigns every arc a positive integer called an arc multiplicity.

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 $\mathfrak{M}(N)$ we denote a set of all markings of a P/T-net N .

We say that transition t in P/T-net $N = (P, T, F, W)$ is *active* in marking M iff for every $p \in \{p \mid (p, t) \in F\}$: $M(p) \geq W(p, t)$. An active transition may *fire*, resulting in a marking M' , such as for all $p \in P$: $M'(p) = M(p) - W(p, t)$ if $p \in \{p \mid (p, t) \in F\}$, $M'(p) = M(p) - W(p, t) + W(t, p)$ if $p \in \{p \mid (t, p) \in F\}$ and $M'(p) = M(p)$ otherwise.

However, for our purpose we use a definition in algebraic representation. Firstly, we define a low-level abstract net

Definition 2. A Low-level Abstract Petri Net is a 4-tuple $(P, T, pre, post)$ where

- P and T are disjoint finite sets of places and transitions, respectively;
- $pre : T \rightarrow N(P)$ is a precondition function;
- $post : T \rightarrow N(P)$ is a postcondition function;

Here, $N : Set \rightarrow Set$ is a functor, defined by $N = G \circ F$, where F is a functor from the category of sets to the category of some structures $Struct$ and G is a forgetful functor from $Struct$ to Set .

Using this concept we can define P/T net as a low-level abstract Petri net where $Struct$ is the category of commutative monoids and F maps each set x to a free monoid $F(x)$ over x ¹.

This definition suggests for a straightforward embedding in Haskell:

```
data Net p t n m = Net
  { places :: Set p
  , trans :: Set t
  , pre :: t -> n p
  , post :: t -> n p
  , initial :: m p
  }
type PNet = Net PTPlace Trans MultiSet MultiSet
type PMark = MultiSet PTPlace
type PTrans = Int
type PTPlace = Int
```

III. NESTED PETRI NETS

In this section we define *nested Petri nets (NP-nets)* [9]. For simplicity we consider here only two-level NP-nets, where net tokens are usual Petri nets.

Definition 3. A nested Petri net is a tuple $(Atom, Expr, Lab, SN, (EN_1, \dots, EN_k))$ where

- $Atom = Var \cup Con$ – a set of atoms;
- Lab is a set of transition labels;
- (EN_1, \dots, EN_k) , where $k \geq 1$ – a finite collection of P/T-nets, called element nets;
- $SN = (P_{SN}, T_{SN}, F_{SN}, v, W, \Lambda)$ is a high-level Petri net where
 - P_{SN} and T_{SN} are disjoint finite sets of system places and system transitions respectively;
 - $F_{SN} \subseteq (P_{SN} \times T_{SN}) \cup (T_{SN} \times P_{SN})$ is a set of system arcs;
 - $v : P_{SN} \rightarrow \{EN_1, \dots, EN_k\} \cup \{\bullet\}$ is a place typing function;
 - $W : F_{SN} \rightarrow Expr$ is an arc labelling function, where $Expr$ is the arc expression language;
 - $\Lambda : T_{SN} \rightarrow Lab \cup \{\tau\}$ is a transition labelling function, τ is the special “silent” label;

The arc expression language $Expr$ is defined as follows.

- Con is a set of constants interpreted over $A = A_{net} \cup \{\bullet\}$ and $A_{net} = \{(EN, m) \mid \exists i = 1, \dots, k : EN = EN_i, m \in \mathfrak{M}(EN_i)\}$, i.e. A_{net} is a set of marked element nets, A is a set of element nets with markings and a regular black token \bullet familiar to us from flat Petri nets (see section above);
- Var is a set of variables, we use variables x, y, z to range over Var .

¹Since there is no commutative monoid datatype in Haskell, we use (isomorphic) representation via multisets.

Definition 4. $Expr$ is a language consisting of multisets over $Con \cup Var$.

The arc labeling function W is restricted in such way that constants or multiple instances of the same variable are not allowed in input arc expressions of the transition, constants and variables in the output arc expressions should correspond to the types of output places, and each variable in an output arc expression of the transition should occur in one of the input arc expressions of the transition.

We use notation like $x + 2y + 3$ to denote multiset $\{x, y, y, \bullet, \bullet, \bullet\}$.

A marking M in an NP-net NPN is a function mapping each $p \in P_{SN}$ to some (possibly empty) multiset $M(p)$ over A .

Let $Vars(e)$ denote a set of variables in an expression $e \in Expr$. For each $t \in T_{SN}$ we define $W(t) = \{W(x, y) \mid (x, y) \in F_{SN} \wedge (x = t \vee y = t)\}$ – all expressions labelling arcs incident to t .

Definition 5. A binding b of a transition t is a function $b : Vars(W(t)) \rightarrow A$, mapping every variable in the t -incident arc expression to some value.

We say that a transition t is *active* w.r.t. a binding b iff

$$\forall p \in \{p \mid (p, t) \in F_{SN}\} : b(W(p, t)) \subseteq M(p)$$

An active transition may fire (denoted $M \xrightarrow{t[b]} M'$) yielding a new marking $M'(p) = M(p) - b(W(p, t)) + b(W(t, p))$ for each $p \in P_{SN}$.

A behavior of an NP-net consists of three kinds of steps: system-autonomous step, element-autonomous step and synchronization step.

- An *element-autonomous step* is a firing of a transition in one of the element nets, which abides standart firing rules for P/T-nets.
- A *system-autonomous step* is a firing of a transition, labeled with τ , in the system net.
- A (*vertical*) *synchronization step* is a simultaneous firing of a transition, labeled with some $\lambda \in Lab$, in a system net together with firings of transitions, also labeled with λ , in all net tokens involved in (i.e. consumed by) this system net transition firing.

IV. USER INTERFACE

The modelling tool of the toolset consists of the meta-model of NP-nets and the tree-based editor which supports editing of NP-nets models. This tool is implemented via well-known modelling framework and code generation facility EMF (Eclipse Modeling Framework). The core of any EMF-based application is the EMF Ecore metamodel which describes domain-specific models. The crucial part of the developed NP-nets metamodel is depicted in fig. 1. The root element of the model is the instance of `PetriNetNestedMarked` class which represents marked NP-nets. `TokenTypeElementNet` class represents element nets. `NetConstant` class represents net constants which bound constants with marked element nets at the

time of NP-net model construction. We omit here the technical details of the remaining part of the metamodel. The metamodel resembles the formal definition of NP-nets given in section III.

The Tree-based editor for the developed metamodel is generated from the Ecore metamodel via EMF codegenerators and modified for the model specific needs. The editor takes care of standard model editing procedures like move, copy, delete, or create fragments of a model and provides undo/redo and serialization/deserialization support.

A NP-net model can be serialized into XMI (XML Meta-data Interchange) representation via the standard serialization mechanism of EMF. Serialized XMI documents are exported to the Haskell backend which carries out analysis procedures.

V. BACKEND

The backend for the tool is written in Haskell [5] and consists of the following parts:

- A library for constructing flat Petri nets;
- A library for constructing nested Petri net;
- Algorithms for checking compositional liveness of nested Petri nets [3];
- A CTL model checker for classical Petri nets;
- Communication layer.

We also make use of a number of GHC extensions which enrich the Haskell's type system.

A. Import

There are two ways to load models into the library: to load the XML file generated by the frontend or to construct the model using specialised library (see section V-B).

For parsing input we use the HXT [6] library based on Arrows [8]. We process the definitions into a `NPNetConstr` code which is later converted to NP-net.

B. Dynamic construction

Libraries for dynamic construction of Petri nets are used in all the other modules of the system. To understand why they are useful, let's take a look at the straightforward definition of a Petri net using the datatype described in the section II.

```
pn1 :: PTNet
pn1 = Net { places = Set.fromList [1,2,3,4]
          , trans = Set.fromList [t1,t2]
          , pre   = \(Trans x) -> case x of
              "t1" -> MSet.fromList [1,2]
              "t2" -> MSet.fromList [1]
          , post  = \(Trans x) -> case x of
              "t1" -> MSet.fromList [3,4]
              "t2" -> MSet.fromList [2]
          , initial = MSet.fromList [1,1,2,2]
          }
  where t1 = Trans "t1"
        t2 = Trans "t2"
```

However, it does get tedious after a while to write out all the nets this way. In addition, such approach is not modular or compositional. We've included a library with simple monadic interface for constructing P/T-nets.

The module `PTConstr` includes a monad `PTConstrM l` which is used for constructing P/T-nets, which transitions

might be labelled with `l`. Among others it also includes the following functions:

```
mkPlace :: PTConstrM l PTPlace
mkTrans :: PTConstrM l PTTrans
label   :: PTTrans -> l -> PTConstrM l ()
```

used for creating places and labelling transitions. In order to have more slick API we use Type Families [1] for providing the interface for arc construction:

```
class Arc k where
  type Co k :: *
  arc :: k -> Co k -> PTConstrM l ()
instance Arc Trans where
  type Co Trans = PTPlace
  arc = ...
instance Arc PTPlace where
  type Co PTPlace = Trans
  arc = ...
```

This allows us to uniformly use `arc` for constructing arcs both from transitions to places and from places to transitions, as shown in the example:

```
pn3 :: PTNet
pn3 = run \$ do
  [t1,t2] <- replicateM 2 mkTrans
  [p1,p2] <- replicateM 2 mkPlace
  label t1 "L1"
  arc p1 t1
  arc p1 t2
  arc t1 p2
  arc t2 p2
```

Furthermore, this allows us to take advantage of type polymorphism and define functions such as

```
arcn :: Arc k => k -> Co k -> Int -> PTConstrM l ()
arcn a b n = replicateM_ n \$ arc a b
```

Similar library for constructing nested Petri nets – `NPNetConstr` – also has facilities for lifting `PTConstrM` code into `NPNetConstrM` monad, which allows for better code reuse.

C. Algorithms

Algorithmically we have implemented a CTL model checker (as shown in [2]) with memoization, algorithm for determining the existence of m -bisimilarity (the algorithm is shown Appendix A) and liveness algorithms (as shown in [4]) which are used for checking liveness in a compositional way.

Definition 6 (Liveness). *A net N is called live if every transition t in its system net is live, eg: $\forall m \in \mathfrak{M}(N). \exists \sigma \in T^*. m \xrightarrow{\sigma} m' \wedge m' \xrightarrow{s} m'' \wedge t \in s$*

Theorem 1. *Let NPN be a marked NP-net with a system net SN and initial marking m_0 . Let also NPN satisfy the following conditions:*

- 1) *(SN, $m_0|_{SN}$) is live (if considered as a separate component);*
- 2) *all net tokens in m_0 and all net constants in every arc expression in NPN are live (if considered as separate components);*
- 3) *for each net token α in m_0 , residing in a place p , α (if considered as a separate component) is m -bisimilar to the α -trail net of p .*

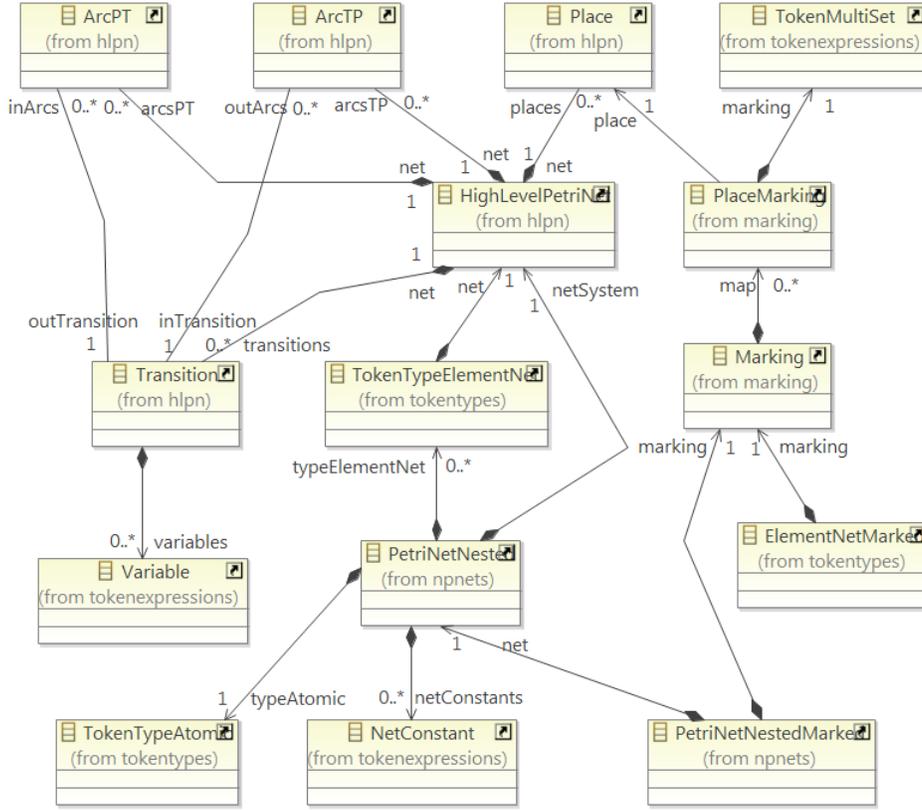


Fig. 1. The EMF Ecore metamodel of NP-nets

Then (NPN, m_0) is live.

For proof of this theorem, definition of α -trail net and algorithm for its construction see [3]. In our project we've implemented the α -trail net construction algorithm and developed the m-bisimilarity checking algorithm (see section A).

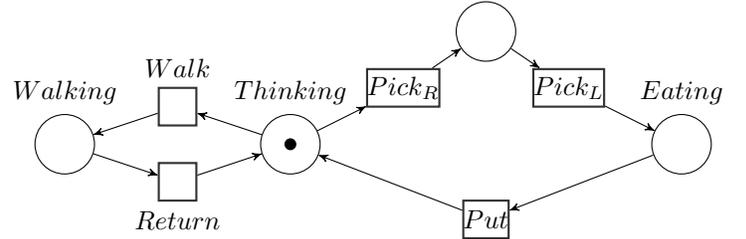


Fig. 2. *philAgent* - A net token representing a single philosopher

VI. EXPERIMENT

For our experiment we decided to check liveness in a compositional way [3] on the following examples: the example net from [3] was checked instantly, due to its facile structure.

We've decided to test our tool on the classical problem of dining philosophers extended with the ability of philosophers to walk: walking philosophers. In our modification philosophers are modeled as separate agents who may exist in different states. Thinking is an important philosophical activity, but who would turn down an opportunity to have a nice walk after a pleasant meal? Therefore philosophers can be either thinking, walking or eating.

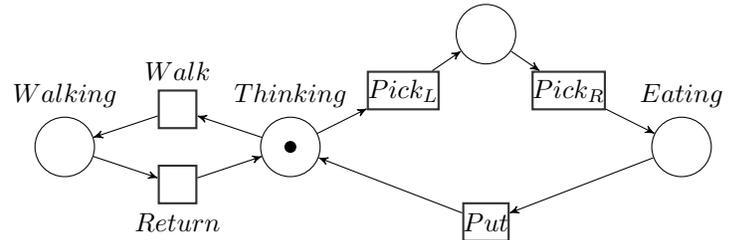


Fig. 3. *lastPhilAgent* - A net token representing the last philosopher

Given a table with n philosophers and n forks, a net, modeling the first $n - 1$ philosophers is shown in figure 2. However, the n -th philosopher is left-handed, and his net is a little bit different (see Fig. 3).

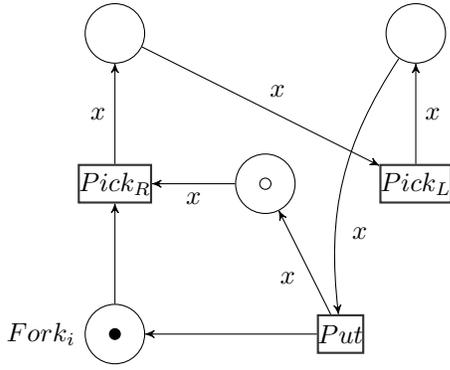


Fig. 4. *phil* - A portion of net representing a philosopher and his right fork

The system net consists of a number of repeated pieces. First $n - 1$ pieces are shown in Fig. 4 and connected in the following way: for each i there is an arc from $Fork_{i+1}$ to $Pick_{R_i}$ and an arc from Put_i to $Fork_{i+1}$. The last piece looks somewhat differently (see Fig. 5) and have arcs from $Fork_1$ to $Pick_{L_n}$ and from Put_n to $Fork_1$.

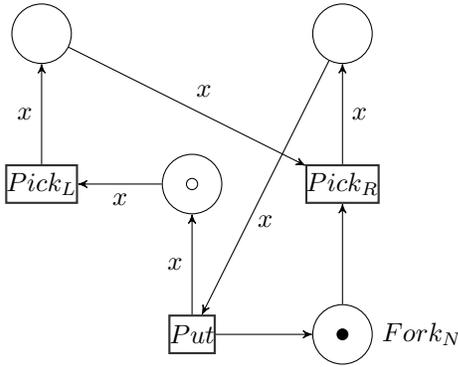


Fig. 5. *lastPhil* - A portion of net representing the last philosopher and his right fork

This system modeled via both interfaces. Firstly the system of 5 philosophers modelled via the frontend modeling tool. We also use API of the backend to automatically generate several system instances with different amount of philosophers and check their liveness.

Due to the modular nature of this task, it was easy to encode it using the construction library from the previous section. The code for the problem is shown in the appendix.

We've verified the compositional liveness of the system for $n = 3, 5, 7, 11$ and got the following results:

Number of philosophers	3	5	7	11
Mean execution time	8.23ms	144.9ms	2.17s	415.5s

The tests were performed using the `criterion` library on the 1.66GHz machine with 993mb RAM running Linux 3.5.0. The data was collected from 20 samples for each test.

VII. CONCLUSIONS AND FURTHER WORK

In this paper we have presented *NPntool* – a support program for nested Petri nets formalism, capable of modeling NP-nets, checking them for liveness in a compositional way, model checking separate components for CTL specifications. We have also developed an algorithm for checking m-bisimilarity needed for liveness. The toolset can be used in both ways - to create and check models with the usage of the NP-nets editor and with the usage of the Haskell-based backend API.

The case study was presented in which we showed how to model NP-nets in a modular way, by modeling a “walking philosophers” problem and testing our tool against it.

Our future works directions includes: implementing a nCTL model checker, implementing a remote simulator. Tree based editor is pretty convenient to create or modify a model, however it is not very helpful to get quick overview of the model or its fragment. So the next step is to implement graphical editor of NP-nets diagrams.

We also intend this tool to be used as a framework for implementing algorithms on nested Petri nets.

APPENDIX A

ALGORITHM FOR CHECKING M-BISIMILARITY

Algorithm 1: *mBisim* – checking for existence of a m-bisimilarity relation

Data: Two nets pt_1, pt_2 with their labelling functions l_1, l_2 and initial markings m_1, m_2 . R of type $\mathfrak{M}(pt_1) \times \mathfrak{M}(pt_2)$ is a relation we are building (initially empty).

Result: *True* if nets are m-bisimilar, *False* otherwise

begin

if $(m_1, m_2) \in R$ **then**

\perp **return** *True*

$T_{s_1} \leftarrow \{t \mid t \in \text{trans}(pt_1) \wedge \text{enabled}(pt_1, m_1, t)\}$

$T_{s_2} \leftarrow \{t \mid t \in \text{trans}(pt_2) \wedge \text{enabled}(pt_2, m_2, t)\}$

 insert (m_1, m_2) in R

for $t \in T_{s_1}$ **do**

$l \leftarrow l_1(t)$

$m'_1 \leftarrow \text{fire}(pt_1, m_1, t)$

$nodes \leftarrow \{n \mid n \in \mathfrak{M}(pt_2) \wedge m_2 \xrightarrow{l} n\}$

if $\text{null}(nodes)$ **then**

\perp **return** *False*

return $\bigwedge \{mBisim(pt_1, pt_2, l_1, l_2, m'_1, m'_2, R) \mid m'_2 \in nodes\}$

for $t \in T_{s_2}$ **do**

$l \leftarrow l_2(t)$

$m'_2 \leftarrow \text{fire}(pt_2, m_2, t)$

$nodes \leftarrow \{n \mid n \in \mathfrak{M}(pt_1) \wedge m_1 \xrightarrow{l} n\}$

if $\text{null}(nodes)$ **then**

\perp **return** *False*

return $\bigwedge \{mBisim(pt_1, pt_2, l_1, l_2, m'_1, m'_2, R) \mid m'_2 \in nodes\}$

The algorithm is implemented using the StateT (Set (PTMark,PTMark)) Maybe monad which allows for a more or less direct translation of the above code.

APPENDIX B WALKING PHILOSOPHERS

```

import NPNTool.PTConstr
import NPNTool.NPNConstr
  (arcExpr, liftPTC, liftElemNet
  , addElemNet, NPNConstrM)
import qualified NPNTool.NPNConstr as NPC

-- Labels
data ForkLabel = PickR | PickL | Put
  deriving (Show,Eq,Ord)

-- Variables
data V = X -- we only need one
  deriving (Show,Eq,Ord)

-- Code for a single philosopher-agent
philAgent :: PTConstrM ForkLabel ()
philAgent = do
  ...

-- Code for the n-th philosopher
lastPhilAgent :: PTConstrM ForkLabel ()
lastPhilAgent = do
  ...

-- returns (Fork_i,PickL_i,Put_i)
phil :: NPNConstrM
  ForkLabel V (PTPlace,Trans,Trans)
phil = do
  [fork,p1,p2,p3] <- replicateM 4 NPC.mkPlace
  [pickL,pickR,put] <- replicateM 3 NPC.mkTrans
  -- get the philAgent token
  agent <- liftElemNet philAgent
  let x = Var X
      NPC.label pickL PickL
      NPC.label pickR PickR
      NPC.label put Put
  -- mark the Fork position with a single token
  NPC.mark fork (Left 1)
  -- mark the philosopher position with an agent
  NPC.mark p1 (Right agent)

  NPC.arc fork pickR
  NPC.arcExpr p1 x pickR
  NPC.arcExpr pickR x p2
  NPC.arcExpr p2 x pickL
  NPC.arcExpr pickL x p3
  NPC.arcExpr p3 x put
  NPC.arcExpr put x p1
  NPC.arc put fork

  return (fork,pickL,put)

lastPhil :: NPNConstrM ForkLabel V (PTPlace,Trans,Trans)
lastPhil = do
  ...

cyclePhils :: Int -> NPNConstrM ForkLabel V ()
cyclePhils n = do
  (fork1,pickL1,put1) <- phil

```

```

(pickL,put) <- midPhils (n-2) (pickL1,put1)
(forkLast,pickLLast,putLast) <- lastPhil
lift $ do
  NPC.arc put forkLast
  NPC.arc forkLast pickL
  NPC.arc fork1 pickLLast
  NPC.arc putLast fork1

midPhils :: Int -> (Trans,Trans)
  -> NPNConstrM ForkLabel V (Trans,Trans)
midPhils n interf | n == 0 = return interf
  | otherwise = do
  (pl,put) <- midPhils (n-1) interf
  (f',pl',put') <- phil
  NPC.arc put f' >> NPC.arc f' pl
  return (pl',put')

diningPhils :: Int -> NPNNet ForkLabel V Int
diningPhils n = NPC.run (cyclePhils n) NPC.new

```

REFERENCES

- [1] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, Simon Marlow, *Associated Types with Class*. – Proceedings of The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), ACM Press, 2005.
- [2] Edmund M. Clarke, Orna Grumberg, Doron Peled. *Model Checking*, MIT Press, 2001.
- [3] L. W. Dworzaski, I. A. Lomazova, *On Compositionality of Boundedness and Liveness for Nested Petri Nets*. – Fundamenta Informaticae, Vol. 120, No. 3-4, pp. 275-293, 2012.
- [4] Serge Haddad, Francois Vernadat, *Analysis Methods for Petri Nets*. – In *Petri Nets: Fundamental Models, Verification and Application*, edited by Michel Diaz, Wiley-ISTE, 656 p., 2009.
- [5] Haskell Programming Language, <http://haskell.org>
- [6] Haskell XML Toolkit, <http://www.fh-wedel.de/~si/HXmlToolbox/index.html>
- [7] K. M. van Hee, I. A. Lomazova, O. Oanea, A. Serebrenik, N. Sidorova, M. Voorhoeve, *Nested nets for adaptive systems*. – Proceedings of the 27th international conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN'06), pp. 241-260, Springer, 2006.
- [8] John Hughes, *Generalising Monads to Arrows*. – Science of Computer Programming 37, 2000.
- [9] I. A. Lomazova, *Nested Petri nets: Modeling and analysis of distributed systems with object structure*. – Moscow:Scientific World, 208 p., 2004.
- [10] I. A. Lomazova, *Nested Petri Nets for Adaptive Process Modeling*. – Lecture Notes in Computer Science, volume 4800, pp. 460-474, Springer, 2008.
- [11] M. Mascheroni, F. Farina, *Nets-Within-Nets paradigm and grid computing*. – Transactions on Petri Nets and Other Models of Concurrency V, pp. 201-220. Springer, Heidelberg, 2012.
- [12] R. Valk, *Petri Nets as Token Objects: An Introduction to Elementary Object Nets*. –ICATPN 1998. LNCS, vol. 1420, pp. 1-25. Springer, Heidelberg, 1998.

The tool for modeling of wireless sensor networks with nested Petri nets

Nina Buchina

Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
m.e.tigra@gmail.com

Leonid Dworzanski

Department of Software Engineering
National Research University Higher School of Economics
Moscow, Russia
leo@mathtech.ru

The work describes a specific approach to modeling and simulating Wireless Sensor Networks (WSN), using nested Petri Nets formalism. The tool for modeling/simulating WSN must take into consideration resources, time and sensors cost. Even though Petri Nets are good for modeling dynamic systems, they do not have enough means to express these aspects. Hence some extension were introduced to handle them.

Keywords—*wireless sensor networks; petri nets; modeling; simulating; analysis;*

I. INTRODUCTION

Two main concepts are discussed in this paper that reader may need to be acquainted with in advance. These are Wireless Sensor Networks and Petri Nets.

A. Wireless sensor networks

Wireless sensor networks are computer networks that consist of distributed sensor nodes. These sensors are used to monitor physical or environmental conditions, such as temperature, sound, pressure and others, and pass the information to the main host (server, end point). These networks are subject of active study, and are utilized in such systems like smart houses, security, environmental and others.

For example, an environmental wireless network described in [3] detects different wild animals as they go through special passages under roads. It also addresses to identification problem. This task is fulfilled by filling the space within some radius from wildlife passages with the wireless nodes of different types and some wireless cameras. In this way scientists may not only find out that the animals have passed the camera, but they also will know the path and direction they followed.

Another widespread usage of WSN are road traffic control and optimization systems. The paper [4] presents a project aimed to “provide a wireless solution for obtaining traffic-related data that can be used for automatically generating safety warnings at black spots along the road network”. The WSN consists of numerous sensors deployed along the road. Each of them monitors its road section, collects parameters such as number, speed and direction of vehicles.

Nets of these types are difficult and expensive to develop and to build. Moreover, the price of possible mistake and error can be extremely high. It may also turn out that the network in principle cannot operate as expected. And as for any complex

system - the sooner this fact will be discovered, the higher are chances to fix the situation before it will drain all the budgets.

This is why modeling and simulating are extremely required for large-scale and expensive networks. There is also a possibility that the deeper understanding of the network will be gained, and the better network structure will be discovered in the process of network modeling.

B. Petri nets

Petri nets is a classical formalism also known as a place/transition net. Petri Nets are commonly used to model different sorts of structures, processes and systems. It also has many extensions as the concept has been adapted for different needs. In this work, two of such extensions will be used. These are Nested Petri nets [1] and Time Petri Nets [2].

It should be outlined that in this work we assume that all the messages in Wireless Sensor Network are intended to end up on the server sooner or later. There may exist specific network that enables sensor nodes to exchange meaningful messages between each other, but by now they are not addressed here.

II. RELATED WORK

A. Wireless sensor network modeling and simulating

There are currently plenty of works concern modeling sensor networks or simulating them.

First of all, there are several mature software products for modeling and simulating networks. Almost all of them share the same drawback: they are very general as they allow to simulate almost any kind of network, and thus are hard to learn and to use for specific needs:

- OPNET [6]
- NetSim [7]
- ns-2 [8]
- OMNet++ [9]

There are also many works performed by researchers, aiming to survey existing network modeling approaches or create new modeling tools.

One of these tools is Shawn [10]. Its authors state that the tool is designed to support several implementation models and propose an algorithm for developing distributed implementation

model of the network. One of main goals of Shawn is to provide support for extremely large networks.

Another wireless sensor network simulating tool that emerged lately is NetTopo [11]. This tool deals with not only WSN model, but also with a real-world wireless sensors testbed. The simulation of virtual WSN, the visualization of real testbed, and the interaction between simulated WSN and testbed are its key challenges.

The review paper [12] summarizes the state of the problem at 2005, which stays mostly relevant.

B. Petri nets and wireless sensor networking

Petri nets are usually taken by some other scientists as a base to create their own modeling language and verification framework.

One of recent works [13] proposes the concept of intelligent wireless sensor networks model (IWSNM) based on Petri nets, which can accurately and unambiguously model the overall and individual characteristics of the networks. Moreover, IWSNM can be analyzed, verified and validated by the supporting tools of Petri nets.

Another work [14] by computer scientists from the University of Virginia proposes an approach to formal application-level requirements specification in wireless sensor networks. They present an event specification language that supports key features of WSNs. As a description language, it is an extension of Petri nets. It integrates features ranging from color, time, and stochastic Petri nets to tackle problems in specification and analysis.

III. GOALS

As it has been shown in the previous section, there are plenty of tools and works intended to help dealing with wireless sensor networks by modeling and simulating them. While commercial products like OPNET can be quite expensive, the academic tools require considerable amount of time and efforts to learn.

On the other hand, a wireless sensor network modeler that is completely free to use and extremely easy to operate would empower more enthusiasts to design their own sensor networks. They would have an opportunity to ensure their projects are actually viable before trying to implement them. Involvement of amateurs may help to accelerate the development of wireless sensor networks industry.

Therefore, the main practical goal of this work is to build a wireless sensor network modeling tool, consisting of 3 modules:

1. Graphical User Interface;
2. The modelling kernel;
3. Analysis backend.

The tool should support some analyzing options such as

1. Liveness
2. Deadlock-freedom
3. Safety

All these are to be done via converting the WSN graph to a Petri net and analyzing this net with standard techniques.

The tool must also be user-friendly and extremely easy to use. It should be possible to perform basic operations even if the user has no deep knowledge on networking or Petri nets.

The theoretical goal of the work is to widen nested Petri notation with WSN-specific features, such as:

1. Time
2. Resources
3. Cost

The last goal is to use the created tool to model and simulate a real wireless sensor network, estimate its characteristics, such as cost, performance etc., and decide whether it can be implemented (and whether it is reasonable).

IV. RESULTS ACHIEVED

A. Going Petri

A conversion scheme was created to translate a set of wireless sensor nodes and their coordinates to the nested Petri net. This scheme consists of 2 parts:

- 1) Generation of overall net model that shows which nodes can communicate to which, and what ways there exist to pass the signal to the server. In general, each sensor node corresponds to two Petri places connected with transition. One of these places is called “implementation place” and is used to store tokens that represent the wireless node implementation. The other place is called a “signal place”. If 2 wireless nodes in real world are able to communicate, their corresponding signal places are connected with Petri transitions like so:

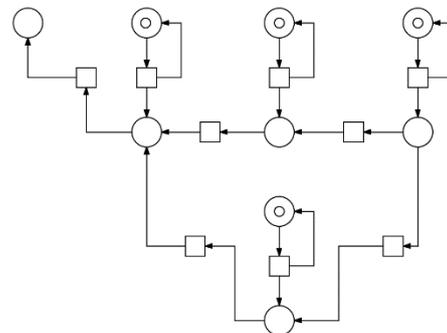


Fig. 1 Sample Petri Net generated from wireless sensor net model. Here each implementation place contains a token.

- 2) Each WS Node has its own Petri model that describes its behavior. It’s network developer’s task to describe the behavior of his wireless nodes. However, the tool will have a bunch of “standard” nodes with predefined behavior. The node’s Petri model is stored in a token that is placed to node’s corresponding implementation Petri place.

The signal is represented by a Petri token. This token travels through the signal states of the net to the server.

Communication is done via horizontal and vertical synchronization [1]. For example, the transition between Node place to its signal place is enabled only when some state of node implementation (stored in token in the implementation place) is achieved.

B. Routing

The order in which nodes participate (or do not) in message transmitting depends on the selected routing algorithm. Currently the tool supports only “least hops” algorithm [5].

The routing is performed on the stage of conversion between WSN structure and Petri net. If two nodes can be used for message transmission, the tool will add a Petri transition to the state that corresponds to the node with least hops number. If there are two nodes with least hops number, the tool will add create both transitions.

This algorithm is about to change when time concept will be introduced. Then no transmission possibilities will be ignored; time windows will be applied for each generated transition instead (see Future work directions, item E: Time problems)

C. The Tool

The tool at its current state allows its user to arrange wireless nodes using simple GUI, and then create a Petri net out of this model with respect to routing.

The tool is written fully on Java and consists of 4 main components.

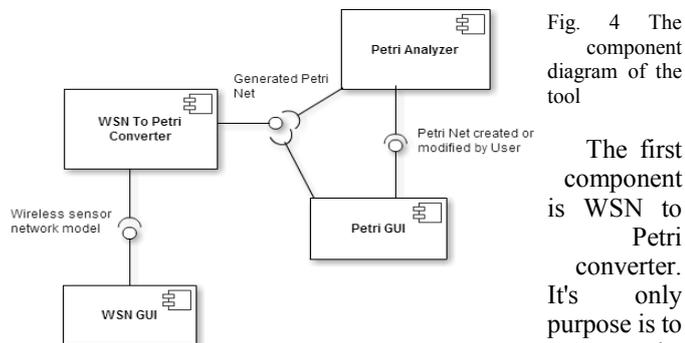


Fig. 4 The component diagram of the tool

The first component is WSN to Petri converter. Its only purpose is to convert the

wireless sensor network model defined by user to its Petri representation with respect to routing algorithm.

The main module of the tool is the Analyzer. It is designed to contain a set of verification algorithms that can be applied to Petri net or Petri-based model. The exact list of algorithms is still being composed, but it is sure to contain tools for analyzing reachability, deadlocks and time characteristics of the net.

The last module is graphical user interface. It actually consists of two main parts.

The first part lets user to arrange wireless sensor nodes on a map or floor plan, which is being uploaded to the tool by the user. The wireless sensor network model is created as a result.

The second part is designed to represent Petri net generated by the tool or created by user. The presented net can also be

modified by the user, except for the case of the Petri Net that was generated from wireless sensor network model. In this case it seems more reasonable to modify the original wireless sensor model and to generate its Petri representation again.

On screen-shots below a sample small wireless sensor network is shown, and so is its Petri representation.

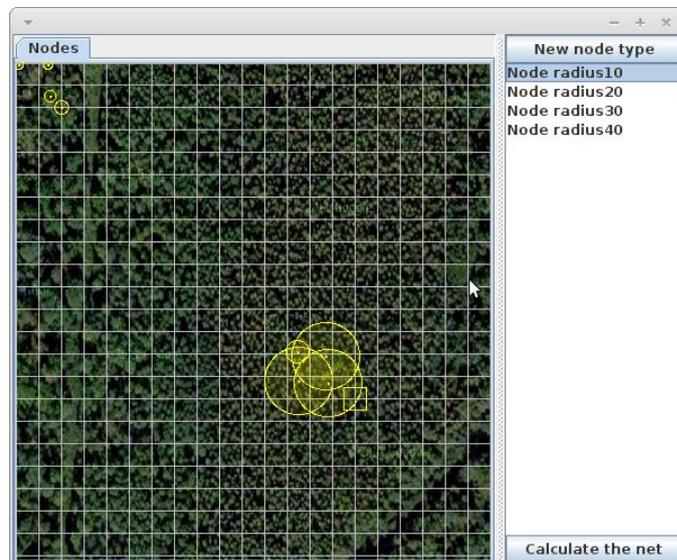


Fig. 2 The Wireless Sensor Network window of the tool.

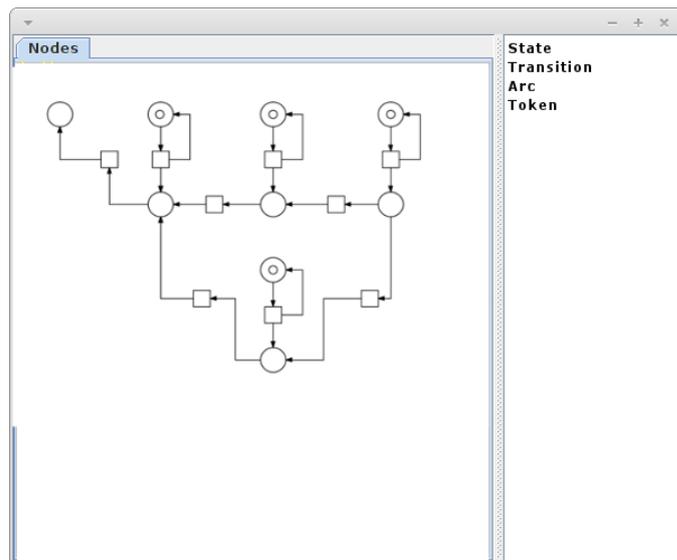


Fig. 3 The Petri Editor window of the tool.

V. FUTURE WORK DIRECTIONS

C. Fully implement the tool

This point assumes implementing all the points from the “Goals” section. It also will require testing the tool against some real WSN and comparing tool’s outputs and prognosis with the real situation.

D. Add support of data packages

At this time, each message is represented by a single token. It would be reasonable to allow user to simulate different message sizes to estimate network bandwidth. For example,

each Petri token can be said to represent 1 KB of data. This work will also require to control each message integrity.

E. Develop more predefined nodes

This point will require creating Petri models of some popular wireless sensor nodes, and introduce these in the tool as representation of wireless sensors behavior. The user of the tool will then be able to edit these models and adapt them for her needs.

F. Analysis options

Some examples of analysis that theoretically can be done using Petri representation of WSN is given in the Goals section. The task is to recognize as many such options as possible and present the results in form of article, also implementing in a tool.

G. Enable the support of decentralized networks

Implementing this feature will allow to model and simulate networks with multiple message end-points (servers), with no servers at all, or with nodes able to consume messages, not just transmit them.

H. Time problems

Introducing the Time to model lets researcher to deeply explore performance of his network. It also lets him to address the problem of network liveness, as the net may reconfigure itself when some nodes become unavailable after certain timeout.

In order to address time problems it has been decided to use Time Petri Net formalism. This concept is described in detail in [2].

This work introduces the concept of time windows. The transition may be fired only if the time elapsed since its last enabling gets into certain interval – the time window. This concept may be used to implement WSN reconfiguration: once the node that is supposed to transmit the message is identified as not responding (the time went out) the net should try to pass the message using a different route.

I. Add multiple routing algorithms support

There are many more possible routing options than just counting hops. The user may prefer routing based on availability, performance, delay time and other characteristics.

The task so far is to add some popular routing algorithms support in a tool. Anyway, it also seems to be reasonable to allow users to create their own routing algorithms. This may be achieved by providing them a java interface to implement and enabling the tool to use these implementations along with build-in routing algorithms.

REFERENCES

[1] И. А. Ломазова “Вложенные сети Петри: моделирование и анализ распределенных систем с объектной структурой”. Научный мир 2003, Москва

[2] L. Popova-Z. “Time and Petri Nets - A Way for Modeling and Verification of Time-dependent Concurrent Systems”. Humboldt-Universität zu Berlin Department of Computer Science December 2012, Moscow

[3] A.-J. Garcia-Sanchez, F. Garcia-Sanchez, F. Losilla, P. Kulakowski, J. Garcia-Haro, A. Rodríguez, J.-V. López-Bao, F. Palomares. “Wireless Sensor Network Deployment for Monitoring Wildlife Passages”. *Sensors* 2010, 10, 7236-7262;

[4] M. Franceschinis, L. Gioanola, M. Messere, R. Tomasi, M. A. Spirito, P. Civera. “Wireless Sensor Networks for Intelligent Transportation Systems”. <http://www-mobile.ecs.soton.ac.uk/home/conference/vtc09spring/DATA/07-02-04.PDF>

[5] Bellman, Richard (1958). “On a routing problem”. *Quarterly of Applied Mathematics* 16: 87–90. MR 0102435

[6] OPNET modeler website: http://www.opnet.com/solutions/network_rd/modeler.html

[7] NetSim website: <http://tetcos.com/>

[8] ns-2 wiki: http://nslam.isi.edu/nslam/index.php/Main_Page

[9] OMNet++ website: <http://www.omnetpp.org/>

[10] Shawn – a customizable sensor network simulator. <https://www.itm.uni-luebeck.de/ShawnWiki/index.php>

[11] Lei Shua, Manfred Hauswirthb, Han-Chieh Chaoc, Min Chend, Yan Zhange: “NetTopo: A framework of simulation and visualization for wireless sensor networks”. *Ad Hoc Networks* 9 (2011) 799–820

[12] E. Egea-López, J. Vales-Alonso, A. S. Martínez-Sala, P. Pavón-Mariño, J. García-Haro: “Simulation Tools for Wireless Sensor Networks”. Summer Simulation Multiconference - SPECTS 2005

[13] X Fu, Z Ma, Z Yu, G Fu: “On Wireless Sensor Networks Formal Modeling Based on Petri Nets”

[14] Binjia Jiao Sang H. Son John A. Stankovic: “GEM: Generic Event Service Middleware for Wireless Sensor Networks”

DPMine: modeling and process mining tool

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—Volume of the data information system operate has been rapidly increasing. Data logs have long been known, as they are a useful tool to solve a range of tasks. The amount of information that is written to a log during a specified length of time leads to the so-called problem of “big data”. Process-aware information systems (PAIS) allow developing models of processes interaction, been monitoring accuracy of their performance and correctness of interaction with each other. Studying logs of PAIS in order to extract knowledge about the processes and construct their models has to do with the process mining discipline. There are available developed tools for process mining, both on a commercial and on a free basis. We are proposing a concept of a new DPMine tool for building a model of multistage process mining from individual processing units connected to each other in a processing graph. The resulting model is executed (simulated) by making an incremental process from the beginning to the end.

Keywords: Process Mining, Model, Tool, Distributing, Workflow, Processes, PAIS

I. INTRODUCTION

As a consequence of information systems (IS) development, volume of the data they operate has rapidly increased. This applies both to the data entered into the system in different ways (automatic, semi-automatic and manual) and the data obtained as a result of some processing which is output by the system to various types of media. In the latter data type one can distinguish a special subclass — the so-called *data log* representing a trace left by some IS and storing information about a set of partial states of the system at different times of its work. These are so-called time logs, which include most of the logs.

Data logs have long been known, as they are a useful tool to solve a range of tasks, including diagnosing errors, documenting a sequence of accessing different nodes of the system, maintaining important system information, etc.

The amount of information that is written to a log during a specified length of time can be quite substantial, making it virtually impossible for a user to manually analyze the user log, which leads to the so-called problem of “big data” [1]. For studying large data represented by some orderly way (e.g. DB), different data-oriented techniques such as machine learning, data mining, etc. are involved.

Of particular interest are process-aware information systems, PAIS, the basic concept of which is the *process* (e.g., a business process or a workflow). These systems allow developing models of processes interaction, monitoring accuracy of their performance and correctness of their interaction with

each other. These include systems such as BPMS (Business Process Management Systems), CRM (Customer Relationship Management), ERP (Enterprise Resource Planning), etc. As in the case with many other IS such systems can produce large logs containing information on interaction of processes in time.

Study of such logs is of great interest from many points of view. For example, this may be a study for obtaining a pattern of processes interaction which reflects the real situation in a subject field in the form of a mathematical and/or graphical model (Petri nets, UML activity diagrams, BPMN, etc.); the so-called *process discovery* problem. The opposite problem is studying compliance of processes execution with an available (developed manually or automatically obtained) model, i.e. *conformance checking* problem. In general, process discovery and conformance checking are related problems. Also, with this model, one can make some adjustments that are designed to show conceptual changes in regard to the subject area, or can perform permanent *enhancement*.

Studying logs of process-aware information systems in order to extract knowledge about the processes and construct their models, as well as studying such models has to do with the *process mining* discipline, which is relevant to data mining, machine learning, process modeling and model-based analysis. Is a relatively young discipline, its goals and objectives are postulated in the *Process Mining Manifesto* [2] supported by more than half a hundred organizations and a large number of experts from various fields [3].

The *event log* is the starting point for process mining research. Typically, such a log is a sequence of *events* united by a common *case* (i.e., a *process instance*). Each event is related to a certain activity which represents a well-defined step in the process. A set of events relating to the same case makes a so-called trace representing an imprint of processes interaction in a particular case.

There is a variety of data mining techniques but most of them employ the following types of information (*resource*): device or person that initiates and then executes an activity, event data elements and event timestamp.

One of the most frequently used forms of Process Mining models representation is *Petri nets* [4]. Petri nets are used both for internal representation of a model by different algorithms [5], [6] and for visualization. Other models, such as heuristic nets and C-nets, are reduced to Petri nets.

To date, there are available developed tools for process mining, both on a commercial and on a free basis. Examples of these tools include *ProM*, which is probably the most widely used process mining tool. ProM is a framework, the possibilities of which are enhanced by plugins, which are several hundreds in number.

Yet for all its power, ProM has a significant architectural constraint: use of the algorithms that are implemented by numerous plugins, can only be done in a discrete mode with direct participation of the user. In other words, one cannot build a chain of processing operations conducting a multistage procedure of log extraction, cannot analyze, transform, and save (or visualize) data derived from such processing.

This paper proposes a concept of a new *DPMine* tool for building a model of multistage process mining from individual processing units (so called *modular blocks*) connected to each other in a *processing graph*. The resulting model is executed (simulated) by making an incremental process from the beginning to the end and producing intermediate and final results.

In the work, by real-world examples we will show differences in approaches to process mining used in tools like ProM and the new DPMine instrument.

The rest of this paper is organized as follows. Section II describes the modular concept and the basic components of DPMine tool. Section III discusses some specific DPMine blocks as applied to some tasks. Certain aspects of practical implementation of the tool are presented in Section IV. Finally, Section V concludes with an analysis of the work done and a look at the future.

II. MODULAR MODEL CONCEPT

Consider the following problem. Suppose there is a log represented in the computer as an XML-file of a specified format. This log should be processed, for this it should be loaded to an appropriate tool that must support this format. An internal representation of the log is formed after downloading the file (partially or fully).

Suppose one wants to create a model as a Petri net, using some of the implemented algorithms, such as conventional x-miner, by executing which the desired Petri net will be formed.

Next, assume that one needs to build a skeleton graph from this Petri net and then to convert it to a passage graph, and to save the results to a file.

To do this using ProM, one should follow these steps:

- 1) Run importing a log file (e.g. in XES format).
- 2) In Actions mode select as Input object "Anonymous log imported from 'file-name.ext'".
- 3) In the list of actions choose the right one for Input object such as "Mine for a Petri Net using Alpha-algorithm".
- 4) Run the action (Petri net construction).
- 5) Set the resulting net as Input Object.
- 6) ...

By consistently performing steps 2–4 for each type of input

object and appropriate action resulting in an output object¹, one can solve the task given above step by step.

Suppose now that after having completed the sequence of actions, the user discovers that at the second stage, while producing a Petri net from the original log, it would have been desirable to use other options of the miner algorithm. This means that all the results obtained after having processed this net become useless and only "obstruct" the resources list. Another example: if a similar sequence of processing operations has to be applied to ten input logs. Or a hundred. Or during mining of the original log one needs to get two models of Petri net, then to compare their similarity.

Obviously, for a flexible solution of such problems a fundamentally different approach to multiprocessing is required, and we proposed it in a tool called DPMine².

Program allows creating a *model of multistep process mining* as a graphical diagram (*processing model*) consisting of *modular blocks*. An example of such a scheme is shown in Fig. 1.

This model defines processing sequence for the initial log (*input object*) present in the form of a file (for example, in XML format). It is then consistently submitted to the mining module, the result in the form of Petri net is replicated by a splitter module and sent in three copies respectively to skeleton module construction, visualization module and the module for saving the Petri net model to a file in one of the available formats.

Blocks are executed sequentially, each block begins its work as it receives input data to its *input port* from the *output port* of the previous block. Depending on block type, for starting block's execution all the data (from all the ports) or partial data may be needed. The modules located "on the right" of the splitter module (Fig. 1) can be executed in parallel, and this can be quasi-parallel in the case of a single processor in a multitasking operating system or "real" parallel if they are run in different threads on a multiprocessor system. There is another version of "true" parallelism: use of special modules for distributed processing that employ as nodes remote computers, which are identified, for example, by IP addresses.

In case of a change in parameters (or input data) of a particular module which is "on the left" in the chain of consecutive executions, the interim data developed "on its right" are announced invalid (invalidation procedure) and are automatically rebuilt (if the model settings are set so). At this, the user does not have to think about how, in what order and at what point he or she needs to perform such a rebuilding: once the scheme is developed, it will then run as many times as it is required to produce a final result satisfying the user.

¹Actually ProM allows manually setting type of output object before step 3, thus narrowing the range of possible actions.

²Letter 'D' in the title has many meanings, among which are such as 'Distinguished', 'Direct' and 'Distributed' (as it will be shown below, one of the key features of the new tool is structural distribution of the original objects for analysis, including that on the set of computers (nodes) for splitting task).

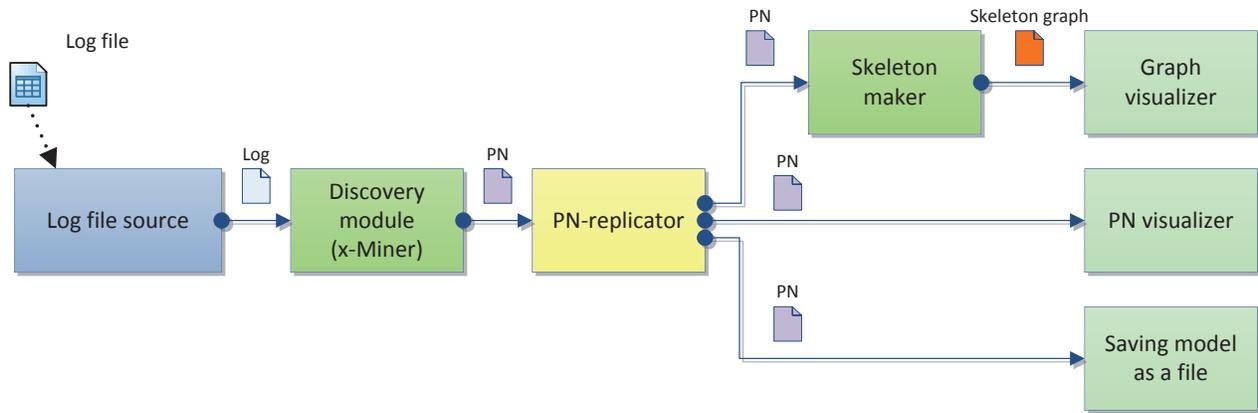


Figure 1. Example of a sequential processing of a log specified in a file

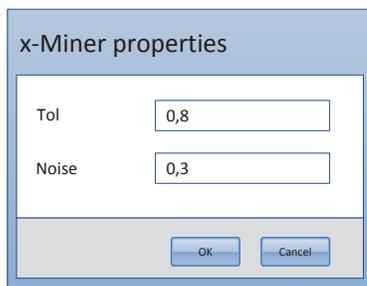


Figure 2. Example of a dialog box for setting parameters of x-miner module

A *processing model* is stored to a file, which allows creating a library of processing models for multiple use for different sets of input data. In addition to information about modules and relationships among them, a processing model stores *modules parameters*, that is a set of characteristics that allow customizing behavior of the modules, for which it is provided. Examples of these parameters can be tolerance params, names of input, output and intermediate files, display options for the modules with a graphical user interface, and so on (see Fig. 2).

As with many other tools, DPMine is developed in a modular fashion that allows extending its functionality through plugins. This way it is possible to connect a large number of third-party modular blocks and significantly expand opportunities for construction of the processing model. Of course, for the entire model to function properly regardless of what modular block is included in its composition, relevant software, implementing their functionality, should be developed in accordance with certain requirements, some of which are covered in the next section.

III. SOME PECULIARITIES OF MODULAR BLOCKS

Connection of modular blocks between each other is made by means of *ports* that can be either *input* or *output* and correspond to the data types they support. Thus, a logical connection (displayed as directed arrows) in the diagram can be achieved only between output (connector start) and the

input (connector end) ports, provided that both ports support the same data type (Fig. 4).

All modular blocks available in DPMine are divided into three groups: *source blocks*, *action blocks* and *render blocks*. They differ in what place in the data processing chain they can occupy.

Consider the purpose of each type of blocks closer.

A. Source blocks

Source blocks have only output ports and, as their name implies, are the source of data to process (Fig. 5). The most obvious example of this block is a log file source that loads a log from the file. Another example would be a block that retrieves a log from some IS, e.g. databases.

Semantically, the source block is a (multi-valued) function with no parameters³ that returns results which can be used by other functions.

B. Render blocks

Render blocks act as consumers of data obtained as a result of processing and have only input ports. By analogy with the source blocks one of the purposes of render blocks is to store results in a file or export to an IS, for example, to a database (Fig. 6).

Along with that, an equally important feature that the render block provides is results visualization. As soon as DPMine is a modeling tool with a user interface, it means that it must support the graphical representation of models in the form of charts, diagrams, graphs, reports, etc. The framework itself does not restrict users to a few possibilities for visualization of results, but instead offers to plug in visualization modules as render-function blocks.

It is a natural question how to proceed in the development of a model when it is necessary both to visualize results and to save them to a file. Here come to rescue special types of blocks — replicator blocks discussed in Section III-E.

³Parameters in this context refers to input and output ports. We should not confuse them with blocks settings, which we shall call *properties*.

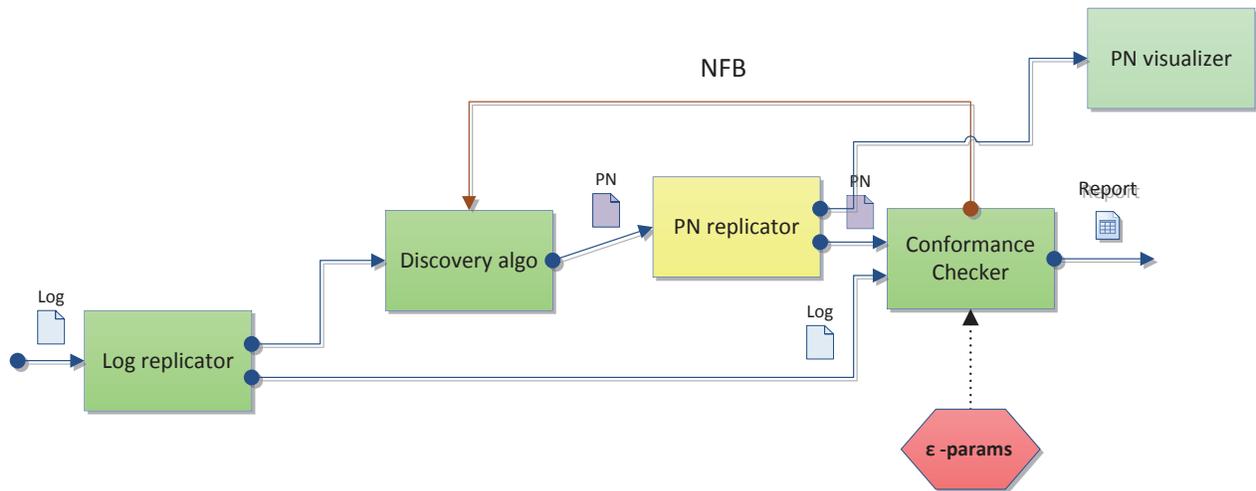


Figure 3. Example of a multistage Process Mining problem solving

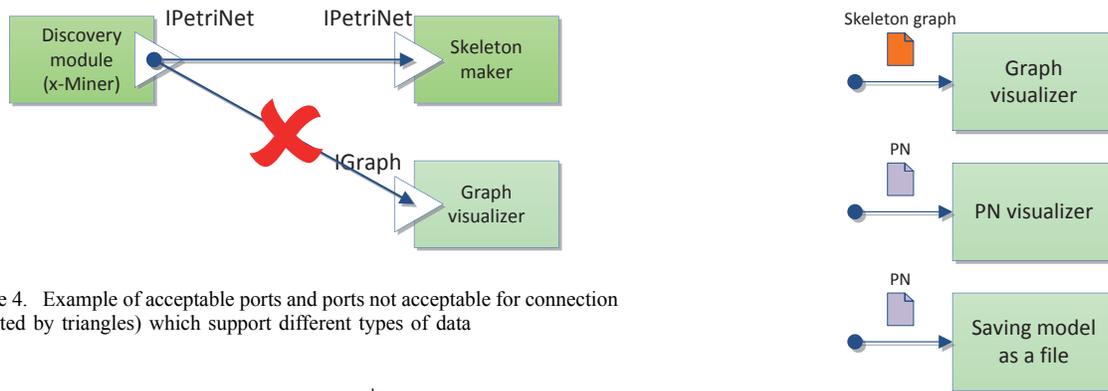


Figure 4. Example of acceptable ports and ports not acceptable for connection (denoted by triangles) which support different types of data

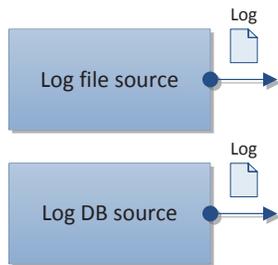


Figure 5. Examples of action blocks

Semantically, the render block is a function of no return values that has results produced by other functions as input parameters.

C. Action blocks

Action blocks are the largest group of blocks having both input and output ports. Semantically, the action block is a function that receives parameters, performs their processing and return them for passing to following functions.

Consider some of the action blocks classes.

Figure 6. Examples of render blocks

D. Action blocks for solving PM tasks

The main purpose of action blocks for solving PM tasks is execution of serial conversion of input data into outputs for solving a particular PM problem. These are block miners for solving Process Discovery task, comparator blocks for solving Conformance Checking, blocks with feedback for Enhancement task (Fig. 7), etc.

There is a variety of blocks in this class that transfer computational load of their tasks to a cloud or any other system of distributed computing grids, etc. (Fig. 8).

Most of the ProM plugins can be attributed to this very class of action blocks.

E. Control action blocks

A separate class includes action blocks not operating data transformation as such but organizing the structure of their execution model. Their purpose in a way is ideologically close to program control command in programming languages. Previously there was cited an example of a problem when the mine-resulting Petri net has to be visualized by a display block

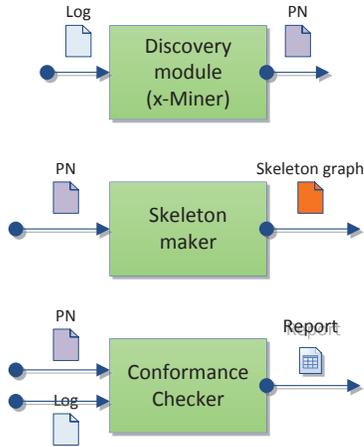


Figure 7. Examples of actions blocks for solving PM tasks

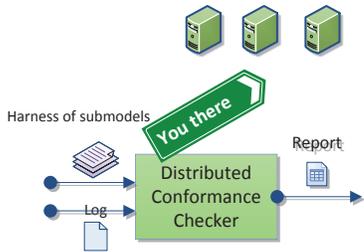


Figure 8. Action block which implements the Conformance Checking task by using distributed calculating nodes

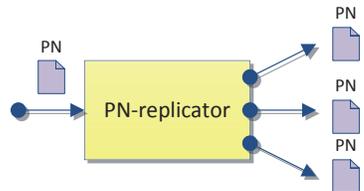


Figure 9. Replicator block performs replication of the inbound PN to multiple outputs

and simultaneously stored (in the form of a model with a set of vertices, transitions, labels, and other support elements) in an external file or database. Here comes to the aid the so-called replicator block (Fig. 9) performing the multiplication of incoming network into multiple outputs. Then one of these outputs can be connected to a visualizer block, another — to a file saving block.

F. Action blocks for distributed calculations

Another major challenge being actively studied in PM is the problem of distributing calculations ([6], [7], [8]). It follows from the computational complexity of the algorithms used in Process Discovery and Conformance Checking, which is manifested in medium and large sets of input data (logs and models), commonly found in experience. It is shown that decomposition of source data (logs section and selection of

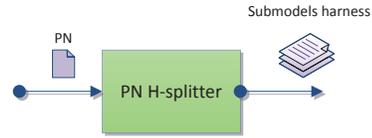


Figure 10. Splitter block with output port in a harness of Petri nets

submodels out of a large model) can significantly reduce the computational load, which stems from the logarithmic complexity of these algorithms.

One can identify three types of distribution: replication, horizontal partitioning of event logs, and vertical partitioning of event logs.

It has been written about replication above, now consider how h-partitioning (h-splitting) and v-partitioning (v-splitting) blocks can be implemented. Based on some heuristic assumptions, the original log is divided into a number of sublogs, which is generally not known in advance. Therefore use of “one sublog — one output port” can be uncomfortable. The answer to this situation is introduction of a special type of data port: *logs harness* and *models harness* (Fig. 10). Of course, to work with a harness, an action block must have an appropriate input port, as in the case with Conformance checking block (Fig. 8). However, one can implement a special demultiplexer block that accepts a harness to input port and its individual elements to outputs ports, and issuing its output ports individual elements of the input harness.

By combining the approaches suggested above, one can build a complex branching model with feedback and get exactly the functionality that is required in each case (Fig. 3).

IV. PRACTICAL IMPLEMENTATION

The last question that to be treated in this paper relates to practical aspects of implementing DPMine tool.

By the time of writing this paper two parallel development works codenamed DPMine/C and DPMine/J have been going on. As the names might suggest, the first fork of the tool is developed mainly in C++ language using Qt 4.8 library as a framework for building GUI, which allows making it cross-platform. Plugin support is performed using dynamic link libraries, for OS Microsoft Windows these are DLLs.

DPMine/J edition is developed in Java, and the main purpose of this line is to attempt to utilize numerous ProM plugins by designing corresponding wrapper classes to use them with DPMine/J framework interfaces. One of GUI options being considered is to utilize IDE Eclipse with a custom plugin implementing the functionality of DPMine tool.

Because at the moment DPMine project is a pure research, probably at some point later a decision will be taken to focus only on one of the forks — either DPMine/C or DPMine/J. Probably the main arguments in favor of the first or second decision will be to what extent it will be possible to utilize ProM plugins and at which cost.

V. CONCLUSION

In this paper DPMine tool concept was discussed. It builds up research models in process mining area. Despite the fact that works on the tool are at the beginning, some of the tasks have already found their realization. Another part is subject to change in the process of working on it.

Among the challenges for the future the following task can be identified, namely developing a declarative language of the underlying graphical model that allows describing complex models.

ACKNOWLEDGMENT

The study was implemented in the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE) in 2013.

The author would like to thank Prof. Irina A. Lomazova for her vital encouragement and support.

REFERENCES

- [1] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers, "Big data: The next frontier for innovation, competition, and productivity," Tech. Rep., 2011.
- [2] IEEE Task Force on Process Mining, "Process Mining Manifesto," in *BPM 2011 Workshops*, ser. Lecture Notes in Business Information Processing, F. Daniel, S. Dustdar, and K. Barkaoui, Eds., vol. 99. Springer-Verlag, Berlin, 2011, pp. 169–194.
- [3] W. M. P. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
- [4] C. A. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, no. 4, p. 6477, 2008.
- [5] J. Carmona, J. Cortadella, and M. Kishinevsky, "A region-based algorithm for discovering petri nets from event logs," in *BPM*, 2008, pp. 358–373.
- [6] W. Aalst, "Decomposing Process Mining Problems Using Passages," in *Applications and Theory of Petri Nets 2012*, ser. Lecture Notes in Computer Science, S. Haddad and L. Pomello, Eds., vol. 7347. Springer-Verlag, Berlin, 2012, pp. 72–91.
- [7] W. M. P. van der Aalst, "Distributed process discovery and conformance checking," in *FASE*, 2012, pp. 1–25.
- [8] —, "Decomposing petri nets for process mining. a generic approach," Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, The Netherlands., Tech. Rep., 2012.

Recognition and Explanation of Incorrect Behavior in Simulation-based Hardware Verification

Mikhail Chupilko*, Alexander Protsenko*†

* Institute for System Programming of the Russian Academy of Sciences (ISPRAS)

† National Research University Higher School of Economics (NRU HSE)

{chupilko,protsenko}@ispras.ru

Abstract—Simulation-based unit-level hardware verification is intended for dynamical checking of hardware designs against their specifications. There are different ways of the specification development and design correctness checking but it is still difficult to diagnose something more than incorrect data on some or other design outputs. The proposed approach is not only to find erroneous design behavior but also to make an explanation of incorrectness on the base of resulted reactions based on special mechanism using a list of explanatory rules.

I. INTRODUCTION

Taking up to 80% of the total verification efforts [1], verification of HDL designs remains being very important. We expect the verification labor costs to be decreased by means of more convenient and substantial diagnostic information. The most complicated problem that underlies in all the approaches to hardware verification is how to represent the specification in machine-readable form that can be both convenient for development and useful for verification purposes. Typically, the specifications can be represented by means of temporal assertions (like in SystemVerilog in general and in Unified verification methodology [2] in particular), or using implicit contracts in form of pre- and post-conditions applied for each operation and micro-operation [3], or by means of executable models. The way of assertion usage lacks of certain incompleteness as assertions covers some of other quality and their possible violation shows only the quality without any guesses why it has happened. To guess something in this case, we should have had a bit higher representation of specifications. The way of implicit specification by means of contracts allows showing which micro operation does not work, but it is still difficult to interpret such information as such interpretation requires lower specification representation.

The executable specification can be considered as being the most useful in the error explanation. To be the most appropriate, the specification should imitate the logic architecture of HDL designs, and the test system is to have mechanisms of explanations the results of simulation. Exactly such mechanisms based on executable specifications are implemented in the proposed approach to test system development as it will be shown later.

The rest of the paper is organized as follows. The following chapter introduces the method of specification and test system development. The third chapter tells about reaction checker work. The fourth chapter reveals the theory underlying the explanatory mechanism. The fifth chapter says a few words about implementation of the approach in C++ library named C++TESK Testing ToolKit [4].

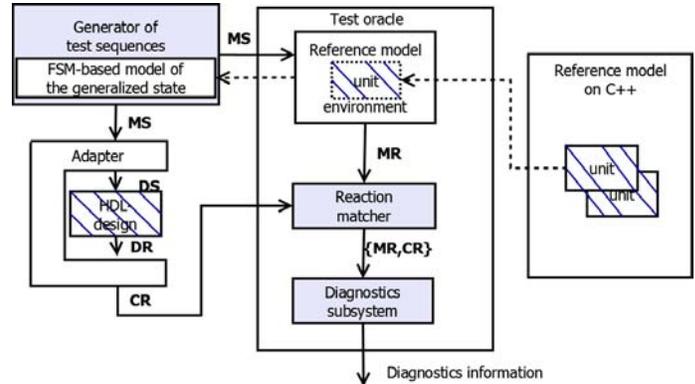


Fig. 1. Common architecture of test system

Then a few words about the approach application are given. The seventh chapter concludes the paper.

II. SPECIFICATION AND TEST SYSTEM ARCHITECTURE

The typical test system for unit-level simulation-based hardware verification includes the following three parts: generator of stimuli, reaction checker, and design under verification (DUV) connected to the test system via special adapter. The proposed approach follows the same tradition but formulates properties of test system components more strictly. Let us shortly consider all the parts of the ordinary test system developed according to the approach (see Figure 1) and then review reference model development more thoroughly.

It should be noticed that fully colored elements in Figure 1 are derived from the supporting library (C++TESK), half-colored elements are developed for each DUV manually on the base of the supporting library, and white-boxed elements are developed fully manually.

Test oracle is the test system core. In fact, the test oracle works as a typical reaction checker; it receives stimuli flow from stimuli generator, receives implementation reactions (DUV reactions) enveloped into messages, and compares them with model reactions produced by reference model. Each message consists of a number of fields carrying data. Only messages with the fields of the same data types are comparable. The test oracle includes a replacement for the reference model which is called reference model environment. The environment consists of a list of operations and functional dependencies between data on output and input interfaces. The operation

description is based on extension of external reference model with timing properties.

The other parts of the test oracle are reaction matcher, and diagnostics subsystem. The reaction sequence made by the reference model is processed by the reaction matcher. It consists of processes each of which processes reactions on one particular reference model interface. As each reference model reaction is bound to a particular output interface, so that all the reactions are subdivided into a set of model interfaces. A reaction arbiter is defined for each output model interface. This component orders model reaction as follows.

When the model reaction is received by the reaction matcher, special process waiting for correspondent implementation reaction is started. If the implementation reaction is found, the process asks the reaction arbiter of the interface whether it can catch the reaction. The reaction arbiter contains a list of model reactions, registered at the interface where the arbiter is defined and not yet matched to the implementation reactions. The match process asking the arbiter about possibility of catching, the arbiter checks the list and according with a strategy of reaction selection (i.e. FIFO, LIFO, data matching) permits or forbids the matching process to catch the implementation reaction.

It is the way of reaction arbitration on each output interface. If the catching is allowed, the model reaction is deleted from the arbiter's reaction list, and the couple of model and implementation reaction is sent to the diagnostics subsystem. If the catching is forbidden, the matching process returns to the state of looking for the next implementation reaction. If the waiting for implementation reaction timeout is reached (the timeout can be set up to each interface separately), the reaction is sent to the diagnostics subsystem alone without implementation reaction marked as *missing reaction*. Besides processes looking for implementation reactions launched by model reactions, special processes named *listeners* are launched by test system for each interface. Each listener is bound to a particular interface and works as follows.

It contains an infinite loop of receiving implementation reaction, shaping the message with the reaction data, checking whether the reaction is matched to the correspondent model reaction at the next cycle after the implementation is completely received by test system. If the matching has happened, the listener returns to its first state and starts looking for the next implementation reaction. If the listener finds out that the implementation reaction has not been taken by any model reactions, it has been waiting for a certain implementation reaction timeout, having placed the implementation reaction into special buffer, offering next model reaction to match with the given implementation reaction. If the implementation reaction timeout is reached, the reaction is sent to the diagnostics subsystem alone without model reaction marked as *unexpected reaction*.

III. REACTION CHECKER ALGORITHM

The reaction checker work can be described by means of an algorithm showing clearly its possibility of catching all visible DUV defects. To provide the algorithm, some introduction might be useful. There are two definitions, the algorithm and a theorem about the reaction checker work.

All the input and output signals of DUV (*implementation*) are subdivided into *input* and *output interfaces*. The set of input and output interfaces of the reference model (*specification*) matches the one of the implementation (*In* and *Out*). Alphabets of stimuli and reactions of the implementation and specification also match each other (X and Y). Set of implementation state (S_{impl}) and specification states (S_{spec}) speaking generally might differ but initial states of implementation and specification are marked out ($s_{impl_0} \in S_{impl}$ and $s_{spec_0} \in S_{spec}$).

Applied during testing to input interface $in \in In$ stimuli are elements of the sequence $\bar{X}^{in} = \langle (x_i, t_i) \rangle_{i=1}^n$, where $x_i \in X$ is a single stimulus, $t_i \in \mathbb{N}_0$ is the time mark of its application ($t_i < t_{i+1}, i = \overline{1, n-1}$). The set of stimuli sequences applied during testing to input interfaces will be denoted as $\bar{X} = \langle \bar{X}^{in_1}, \dots, \bar{X}^{in_n} \rangle$ and called *stimuli sequence*. Stimuli sequence admissibility is defined by definitional domain $Dom \subseteq \bigcup_{k=0}^{\infty} (X \times \mathbb{N}_0)^k$.

Implementation answering the stimuli sequence \bar{X} produces reactions $\bar{Y}_{impl}^{out}(\bar{X}) = \langle (y'_i, t'_i) \rangle_{i=1}^m$ and sends them to the output interface $out \in Out$, where $y'_i \in Y$ is a single reaction, $t'_i \in \mathbb{N}_0$ is time of its sending ($t'_i < t'_{i+1}, i = \overline{1, m-1}$). Let the set of reaction sequences emitting by the implementation to all interfaces be denoted as $\bar{Y}_{impl} = \langle \bar{Y}_{impl}^{out_1}, \dots, \bar{Y}_{impl}^{out_M} \rangle$ and called *implementation reaction sequence*.

Specification answering the stimuli sequence \bar{X} produces reactions $\bar{Y}_{spec}^{out}(\bar{X}) = \langle (y_i, t_i) \rangle_{i=1}^k$ and sends them to the output interface $out \in Out$, where $y_i \in Y$ is a single reaction, $t_i \in \mathbb{N}_0$ is time of its sending ($t_i \leq t_{i+1}, i = \overline{1, k-1}$). Let the set of reaction sequences emitting by the implementation to all interfaces be denoted as $\bar{Y}_{spec} = \langle \bar{Y}_{spec}^{out_1}, \dots, \bar{Y}_{spec}^{out_K} \rangle$ and called *specification reaction sequence*.

Let each output interface $out \in Out$ to be equipped with reaction production timeout $\Delta t^{out} \in \mathbb{N}_0$. Let also each finite stimuli sequence results in a finite reaction sequence. Let us denote single element of reaction sequence $\bar{Y} = \langle (y_i, t_i) \rangle_{i=1}^k$ as $\bar{Y}[i] = (y_i, t_i)$. The operation of element removing from the reaction sequence $\bar{Y} \setminus (y, t)$ is defined as follows: if the element being removed is absent in the sequence, the result consists of the former sequence; if the element is in the sequence, its first entrance in the sequence will be removed. The sequence length is denoted as $m = |\bar{Y}|$.

Definition 1: The implementation is said to correspond to the specification if $\forall out \in Out$ and $\forall \bar{X} \in Dom$ $|\bar{Y}_{impl}^{out}(\bar{X})| = |\bar{Y}_{spec}^{out}(\bar{X})| = m^{out}$ is satisfied and there is a rearrangement π^{out} of the set $\{1, \dots, m^{out}\}$ so that $\forall i \in \{1, \dots, m^{out}\}$ $\left\{ \begin{array}{l} y'_i = y_j \\ t_j \leq t'_i \leq t_j + \Delta t^{out} \end{array} \right\}$ is satisfied, where $j = \pi^{out}(i)$.

Definition 2: The implementation behavior is said to have an observable failure if the implementation does not correspond to the specification or $\exists \bar{X} \in Dom$ and $\exists out \in Out$ so that either $|\bar{Y}_{impl}^{out}(\bar{X})| \neq |\bar{Y}_{spec}^{out}(\bar{X})|$, or for each rearrangement π^{out} of the set $\{1, \dots, m^{out}\}$ $\exists i \in \{1, \dots, m^{out}\}$ for which $\left[\begin{array}{l} y'_i \neq y_j \\ t_j > t'_i \\ t'_i > t_j + \Delta t^{out} \end{array} \right]$ is satisfied, where $j = \pi^{out}(i)$.

Action 1 *reactionMatcher* $[\bar{Y}_{impl}, \bar{Y}_{spec}]$

Guard: *true*

Input: $\bar{Y}_{impl}, \bar{Y}_{spec}$

$\bar{Y}_{spec}^* \leftarrow \bar{Y}_{spec}$

for all $i \in |\bar{Y}_{impl}|$ **do**

$t^{now} \leftarrow t'_i$

$Y_{spec}^{now} \leftarrow \{(y, t) | \exists j \cdot (y, t) = \bar{Y}_{spec}^*[j] \wedge t \leq t^{now}\}$

$Y_{missing}^{now} \leftarrow \{(y, t) \in Y_{spec}^{now} | t + \Delta t < t^{now}\}$

if $Y_{missing}^{now} \neq \emptyset$ **then**

$Y_{defect} \leftarrow Y_{defect} \cup Y_{missing}^{now}$

end if

$Y_{matched}^{now} \leftarrow \{(y, t) \in Y_{spec}^{now} | y = y'_i \wedge t \leq t'_i \leq t + \Delta t\}$

if $Y_{matched}^{now} = \emptyset$ **then**

$Y_{defect} \leftarrow Y_{defect} \cup (y'_i, t'_i)$

end if

$(y_{matched}, t_{matched}) \leftarrow \underset{(y,t) \in Y_{matched}^{now}}{\operatorname{argmin}} t$

$\bar{Y}_{spec}^* \leftarrow \bar{Y}_{spec}^* \setminus (y_{matched}, t_{matched})$

end for

if $|\bar{Y}_{spec}^*| \neq 0$ **then**

$Y_{defect} \leftarrow Y_{defect} \cup \bar{Y}_{spec}^*$

end if

return Y_{defect}

Lemma 1: If reaction sequence \bar{Y}_{impl} and \bar{Y}_{spec} are finite, and $|\bar{Y}_{impl}| \neq |\bar{Y}_{spec}|$ then test oracle returns negative verdict.

Proof: Suppose the main cycle of the algorithm not to find a failure. In this case the number of elements in sequence \bar{Y}_{spec}^* (which at the first step was equal to the number of elements in \bar{Y}_{spec}) will be decreased to the number, which the sequence \bar{Y}_{impl} contains. If $|\bar{Y}_{impl}| > |\bar{Y}_{spec}|$, then there is no step of the test oracle algorithm to find reaction from sequence \bar{Y}_{spec} correspondent to current being worked under reaction from sequence \bar{Y}_{impl} . In this case test oracle finishes its work with negative verdict. We had supposed that such a situation cant occur, so that $|\bar{Y}_{impl}| < |\bar{Y}_{spec}|$. In this case $|\bar{Y}_{spec}^*| = |\bar{Y}_{spec}| - |\bar{Y}_{impl}| > 0$ and the oracle finishes its work with negative verdict in due to condition *if* $|\bar{Y}_{spec}^*| \neq \emptyset$ *then return (false)* after the main cycle having finished. ■

Theorem 1: Test oracle working according to the proposed algorithm allows constructing significant tests (it means that oracle is not mistaken having found certain defect).

Proof: The case when $|\bar{Y}_{impl}| \neq |\bar{Y}_{spec}|$ meaning that there are different numbers of implementation and specification reactions is considered to be erroneous according to the definition 2. It was considered in the lemma and shown that the test oracle in this case does return negative verdict.

Let us consider the case $|\bar{Y}_{impl}| = |\bar{Y}_{spec}| = 0$. Here the main cycle of test oracle work is not executed, the condition *if* $|\bar{Y}_{spec}^*| \neq \emptyset$ *then return (false)* is not satisfied too and the test oracle returns positive verdict (true). The case of empty sequences is understood as correct according to the definition 2. According to the induction rule of inference, let us suppose that for the case $|\bar{Y}_{impl}| = |\bar{Y}_{spec}| = n$ test oracle returns verdict correctly. Let us prove that the same situation takes place if the numbers of elements in sequences are equal to $n + 1$. According to the definition 2, defect can be found if for each rearrangement π of set $1, \dots, n \exists i \in 1, \dots, n$, when

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	$(r_{spec}, NULL)$	$\nexists r_{impl} \in R_{impl} \setminus R_{impl}^{normal, incorrect} : iface_{spec} = iface_{impl} \ \& \ time_{min} < time < time_{max}$
UNEXPECTED	$(NULL, r_{impl})$	$\nexists r_{spec} \in R_{spec} \setminus R_{spec}^{normal, incorrect} : iface_{impl} = iface_{spec} \ \& \ time_{min} < time < time_{max}$

TABLE I. REACTION CHECKER REACTION PAIR TYPES

$\left[\begin{array}{l} y'_i \neq y_j \\ t_j > t'_i \\ t'_i > t_j + \Delta t^{out} \end{array} \right]$ is satisfied, where $j = \pi(i)$.

Let us remove last elements of sequences and make sequences where the numbers of elements are equal to n and to which test oracle works correctly. Let us consider the case of the following two removed reactions. Negative verdict can be returned only in two cases: the first one is *if* $Y_{missing}^{now} \neq \emptyset$ *then return (false)*, the second one is *if* $Y_{matched}^{now} = \emptyset$ *then return (false)*. The first case occurs only when $t'_i > t_j + \Delta t^{out}$ according to the definition 2, and the second case takes place only in when $\left[\begin{array}{l} y'_i \neq y_j \\ t_j > t'_i \end{array} \right]$ according to the definition 2. Therefore, test oracle returns negative verdict only when there is erroneous reaction in any finite reaction sequences. ■

IV. DIAGNOSTICS SUBSYSTEM

Let 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 timemark: generation of implementation reaction always has concrete time mark.

The reaction checker has already attempted to match each reaction from R_{spec} with a reaction from R_{impl} , making a *reaction pair*. 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 *normal, missing, unexpected, 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 simultaneously be located only in one pair.

The diagnostics subsystem has its own interpretation of reaction pair types (see Table II). In fact, the subsystem translates original reaction pairs received from the reaction checker into new representation. This process can be described as $M \Rightarrow M^*$, where $M = \{(r_{spec}, r_{impl}, type)_i\}$ is a

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

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 these might be different M^* dependent on the algorithm of its creation (accounting for original order, strategy of reaction pair selection for recombination, etc). This question will be discussed after so called *transformation rules* are presented.

Having made reaction pair set, reaction matcher sends it to the *diagnostics subsystem* to process them providing verification engineers with explanation of problems having occurred in the verification process. The diagnostics subsystem is underlain with a special algorithm, consisting of consequent application of the set of so called *rules*, each of which transforms the reaction pairs. Some rules decrease the number of pairs, having found pairs with correspondent implementation and specification reactions, collapse them and write diagnostics information into log-file. Other rules make it possible to recombine reaction pairs for better application of rules from the first type. The third part of rules uses special technique to find similar reactions according to the *distant function* to recombine the reaction pairs for better readability but do. The distant function can be implemented in three possible ways. To begin with, it may account the number of equal data fields in two given messages. Second, Hamming distance may be used as one can compare not only the fields but the bits of data carried by the fields. The measure of closeness between two given reactions is denoted as $C(r_{spec}, r_{impl})$.

Each rule consists of one or several pairs of reactions. In cases of missing of unexpected reactions, one of the pair elements is undefined and called *null*. Each pair of reaction is assigned with model interface. Left part of the rule shows initial state and right part (after the arrow) shows result of the rule application. If the rule is applied to several reaction pairs, they are separated with comma. Now, let us review all these twelve rules that we found.

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 $(a_{spec}, b_{impl}), (b_{spec}, a_{impl}) : data_{a_{spec}} = data_{a_{impl}} \ \& \ data_{b_{spec}} = data_{b_{impl}}$, these reaction pairs should be *regrouped*. $\{(a_{spec}, b_{impl}), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, b_{impl})\}$.

Rule 4: If there is a missing reaction pair and an unexpected reaction pair $(a_{spec}, null), (null, a_{impl}) : data_{a_{spec}} = data_{a_{impl}}$, they should be united into one reaction pair. $\{(a_{spec}, null), (null, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl})\}$.

Rule 5: If there is a missing reaction pair and an incorrect reaction pair $(a_{spec}, null), (b_{spec}, a_{impl}) : data_{a_{spec}} = data_{a_{impl}}$, these reaction pairs should be *regrouped*. $\{(a_{spec}, null), (b_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (b_{spec}, null)\}$.

Rule 6: If there is an unexpected reaction pair and an incorrect reaction pair $(null, a_{impl}), (a_{spec}, b_{impl}) : data_{a_{spec}} = data_{a_{impl}}$, these reaction pairs should be *regrouped*. $\{(null, a_{impl}), (a_{spec}, b_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (null, b_{impl})\}$.

Rule 7: If there are two incorrect reaction pairs $(a_{spec}, b_{impl}), (c_{spec}, a_{impl}) : data_{a_{spec}} = data_{a_{impl}}$, these reaction pairs should be *regrouped*. $\{(a_{spec}, b_{impl}), (c_{spec}, a_{impl})\} \Rightarrow \{(a_{spec}, a_{impl}), (c_{spec}, b_{impl})\}$.

The rules 1-7 allow finding the closest reaction pairs. The algorithm of their implementation is shown in 2 and 4 algorithms.

Action 2 *match*[($r_{1_{spec}}, r_{1_{impl}}$), ($r_{2_{spec}}, r_{2_{impl}}$)]

Input: $RP_1 = (r_{1_{spec}}, r_{1_{impl}}), RP_2 = (r_{2_{spec}}, r_{2_{impl}})$
for all *rule_number* $\in |NormalRules|$ **do**
 if *rules*[*rule_number*].*isApplicable*(RP_1, RP_2) **then**
 return *rule_number*
 end if
end for
return 0

Action 3 *fuzzy_match*[($r_{1_{spec}}, r_{1_{impl}}$), ($r_{2_{spec}}, r_{2_{impl}}$)], *rule*]

Input: $RP_1 = (r_{1_{spec}}, r_{1_{impl}}), RP_2 = (r_{2_{spec}}, r_{2_{impl}})$
proximity_metric $\Leftarrow 0$
for all *rule_number* $\in |FuzzyRules|$ **do**
 if (*metric** = *rules*[*rule_number*].*metric*(RP_1, RP_2)) > *proximity_metric* **then**
 proximity_metric \Leftarrow *metric**
 rule \Leftarrow *rule_number*
 end if
end for
return *proximity_metric*

Action 4 *apply_normal_rules*[\{(r_{spec}, r_{impl})_{*i*}\}]

Input: \{(r_{spec}, r_{impl})_{*i*}\}
for all *r* $\in |\{(r_{spec}, r_{impl})_i\}|$ **do**
 if !*r.collapsed* **then**
 for all *p* $\in |\{(r_{spec}, r_{impl})_i\}|$ **do**
 if !*r.collapsed* & !*p.collapsed* **then**
 if *rule_number* = *match*(*r, p*) **then**
 ($r_{spec_{i+1}}, r_{impl_{i+1}}), (r_{spec_{i+2}}, r_{impl_{i+2}})$ \Leftarrow
 rules[*rule_number*].*apply_rule*(*r, p*)
 r.collapsed \Leftarrow *true*
 p.collapsed \Leftarrow *true*
 return
 end if
 end if
 end for
 end if
end for

Action 5 *apply_fuzzy_rules*[\{(r_{spec}, r_{impl})_i\}]

Input: \{(r_{spec}, r_{impl})_i\}

for all r ∈ |\{(r_{spec}, r_{impl})_i\}| **do**

if !r.collapsed **then**

 metric* ← 0

for all p ∈ |\{(r_{spec}, r_{impl})_i\}| **do**

if !r.collapsed & !p.collapsed **then**

 metric = fuzzy_match(r, p, rule_number)

if metric > metric* **then**

 metric* ← metric

 rule_number* ← rule_number

 s₁ ← r

 s₂ ← p

end if

end if

end for

if metric* > 0 **then**

 (r_{spec_{i+1}}, r_{impl_{i+1}}), (r_{spec_{i+2}}, r_{impl_{i+2}}) ←

 rules[rule_number*].apply_rule(s₁, s₂)

 s₁.collapsed ← true

 s₂.collapsed ← true

return

end if

end for

When the rules from the list of normal rules have been applied, the sets R_{spec} and R_{impl} does not contain any not yet collapsed reactions with identical data. In this part of diagnostics subsystem work the stage of *fuzzy rules* (See 3 and 5 algorithms) comes.

Rule 8: If there are two reaction pairs $\{(a_{spec}, b_{impl}), (b'_{spec}, a'_{impl})\}$: $c(a_{spec}, a'_{impl}) < c(a_{spec}, b_{impl})$ & $c(a_{spec}, a'_{impl}) < c(b'_{spec}, a'_{impl})$ or $c(b'_{spec}, b_{impl}) < c(a_{spec}, b_{impl})$ & $c(b'_{spec}, b_{impl}) < c(b'_{spec}, a'_{impl})$, where c is the selected distance function and the value of c is the best among other fuzzy rules, these reaction pairs should be regrouped. $\{(a_{spec}, b_{impl}), (b'_{spec}, a'_{impl})\} \Rightarrow \{(a_{spec}, a'_{impl}), (b'_{spec}, b_{impl})\}$

Rule 9: If there are two reaction pairs $\{(a_{spec}, null), (null, a'_{impl})\}$ and the value of the selected distant function $c = c(a_{spec}, a'_{impl})$ is the best among other fuzzy rules, these reaction pairs should be regrouped. $\{(a_{spec}, null), (null, a'_{impl})\} \Rightarrow \{(a_{spec}, a'_{impl})\}$

Rule 10: If there are two reaction pairs $\{(a_{spec}, null), (b_{spec}, a'_{impl})\}$: $c(a_{spec}, a'_{impl}) < c(b_{spec}, a'_{impl})$, where c is the selected distance function and the value of c is the best among other fuzzy rules, these reaction pairs should be regrouped. $\{(a_{spec}, null), (b_{spec}, a'_{impl})\} \Rightarrow \{(a_{spec}, a'_{impl}), (b_{spec}, null)\}$

Rule 11: If there are two reaction pairs $\{(null, a_{impl}), (a'_{spec}, b_{impl})\}$: $c(a'_{spec}, a_{impl}) < c(a'_{spec}, b_{impl})$, where c is the selected distance function and the value of c is the best among other fuzzy rules, these reaction pairs should be

regrouped. $\{(null, a_{impl}, null), (a'_{spec}, b_{impl})\} \Rightarrow \{(a'_{spec}, a_{impl}), (null, b_{impl})\}$

When all the metrics of fuzzy rules have been measured and all the most suitable rules have been applied, the time of the last rule comes.

Rule 12: If there is a reaction pair (a_{spec}, a'_{impl}) with both specification and implementation parts, it should be collapsed. $(a_{spec}, a'_{impl}) \Rightarrow (null, null)$.

The last rule allows transforming all the incorrect reaction pairs to show the diagnostics for the whole list of reaction pairs. Typically, after the application of each rule, the history of transformation is traced and then it is possible to reconstruct the parents of the given reaction pairs and all the rules they are undergone. Such a reconstruction of the rule application trace we understand as the *diagnostics information*.

V. IMPLEMENTATION

The proposed approach to development of test systems, reference model construction, reaction correctness checking, and diagnostics subsystem has been implemented in the open source library C++TESK Testing ToolKit [4] developed by ISPRAS. The library is developed in C++ language to be convenient for verification engineers. It contains macros enabling the engineers to develop all the parts of the test systems which should be done by hands. Some parts, like diagnostics subsystem algorithm, are hidden inside of the tool.

Results of diagnostics work are shown each time after the verification is over. Now they look like tables with all found errors and results of rule application: new reaction pair sets and the way of their obtaining.

VI. RESULTS

The C++TESK testing toolkit including diagnostics subsystem has been used in the number of projects of industrial microprocessor development in Russia. The aim of the all approach is unit-level verification and on this level it can be a competitor to widely used UVM mentioned in the introduction.

It might be shown by the following fact. Typically, we started verification by means of C++TESK starts when the whole system had been already verified by UVM-like approaches. In spite of power of UVM, it does not include means to direct test sequence generation, which C++TESK does, means of quick analysis of verification results as diagnostics subsystem etc.

Results of application of different approaches depend on the qualification of the engineers and their familiarity with the approach. And on this point, we should say that our toolkit was used by people now being close to its development kitchen and despite it, they exactly managed to find those bugs we have already mentioned.

VII. CONCLUSION

The proposed approach to simulation-based unit-level hardware verification solves in some sense the task of dynamical checking of hardware designs against their specifications. It includes both means of specification development and diagnostics subsystem producing an explanation of incorrectness

on the base of special mechanism using formally represented specifications and a list of explanatory rules.

The approach has been used in the number of projects and shown its possibility to find defects and help verification engineers to correct them by means of diagnostics information.

Our future research is connected with more convenient representation of diagnostics results by means of wave-diagrams, localization of found problems in source-code.

REFERENCES

- [1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Pub, 2003.
- [2] Unified verification methodology. [Online]. Available: <http://www.uvmworld.org>
- [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/>

Horizontal Transformations of Visual Models in MetaLanguage System

Alexander O. Sukhov

Department of Software and Computing Systems
Mathematical Support
Perm State University
Perm, Russian Federation
E-mail: Sukhov.psu@gmail.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. Different specialists are involved in software development at once: databases designers, business analysts, user interface designers, programmers, testers, etc. It leads to creation and usage in systems designing of various models fulfilled from the different points of view, with different levels of details, which use different modeling languages for the description. Thus there is a necessity of models transformation as between different levels of hierarchy, and within the same level between different modeling languages for creation of united model of system and exporting of models to external systems. The MetaLanguage system is intended to visual domain-specific languages creation. The approaches to development of a model transformation component of MetaLanguage system are considered. This component allows to fulfill vertical and horizontal model transformations of “model-text” and “model-model” types. These transformations are based on graph grammars described by production rules. Each rule contains the left- and right-hand sides. The algorithm of the left-hand side search in the source model and the algorithms of execution of a right-hand side of a rule are described. Transformations definitions for models in ERD notation are presented as example.

Keywords: *model-based approach; visual model; domain-specific language; horizontal model transformation; language workbench.*

I. INTRODUCTION

In industrial production we often come to the fact that the studying and creation of an object is done by constructing its model. Since development of computer systems the idea of creation and usage of models has come to computer science.

Model is an abstract description of system (object, process) which contains characteristics and features of its functioning which are important from the viewpoint of modeling purposes. *Metamodel* is a language used for models development. For metamodels description the *meta-metamodels (metalanguages)* are used. *Modeling* is the process of creation and studying of models.

Today the majority supposes that visual models are used only at the early stages of software development, for creation of certain “sketch” of system or transfer of high-level ideas of

designing, i.e. it is supposed that models play a secondary role, and are primarily used only for documentation. However there are approaches to system engineering in which the basic elements are the visual models and their transformations – model-based approaches.

The model-based approaches are capable at information system creation to unite efforts of developers and domain experts. These approaches make the system more flexible, since for its change there is no necessity of modification of source code “by hand”, it is enough to modify a visual model, and with this task even nonprofessional programmers can cope.

For model-based approaches implementation it is necessary to use toolkit which will be convenient to various participants of system development process. The general-purpose modeling languages, such as UML, are not able to cope with this task, because they have some disadvantages:

- Diagrams are complicated for understanding not only for experts, who take part in system engineering, but in some cases even for professional developers.
- Object-oriented diagrams can not adequately represent domain concepts, since work is being done in terms of “class”, “association”, “aggregation”, etc., rather than in domain terms.

That is why at implementation of model-based approaches the domain-specific modeling languages (DSMLs, DSLs), created to work in specific domains, are increasingly used. Domain-specific languages are more expressive, simple on applying and easy to understand for different categories of users as they operate with domain terms. Therefore now a large number of DSLs is developed for using in different domains [1-3].

Despite all DSLs advantages they have one big disadvantage – complexity of the designing. If general purpose languages allow creating programs irrespectively to domain, in case of DSLs for each domain, and in some cases for each task it is necessary to create the domain-specific language. Another shortcoming of domain-specific language is that it is necessary to create convenient graphical editors to work with it.

The *language workbench* or *DSM-platform* is the instrumental software intended to support development and maintenance of DSLs. Usage at DSLs creation a language workbench considerably simplifies the process of their designing. The *MetaLanguage* [4] system is a language workbench for creating visual dynamic adaptable domain-specific modeling languages. This system allows fulfilling multilevel and multi-language modeling of domain.

The different categories of users work at various stages of system life cycle. At the stage of system creation the leading role is played by professional developers with participation of experts, specialists in the appropriate domain, and at the operation stage – by experts, specialists and end-users, as they detect all system shortcomings and mistakes in its implementation. To attract experts and specialists to the process of system adjustment of the ever-changing operating conditions and user requirements it is necessary to provide them with the convenient language, which is operates with customary terms. Using this language they could make all necessary modifications of information system.

On the other hand, several specialists are involved in software development at once: databases designers, business analysts, user interface designers, programmers, testers, etc. Each of these specialists uses their own information about the system and this information may describe the same objects, but from the different points of view and with various modeling languages (see fig. 1).

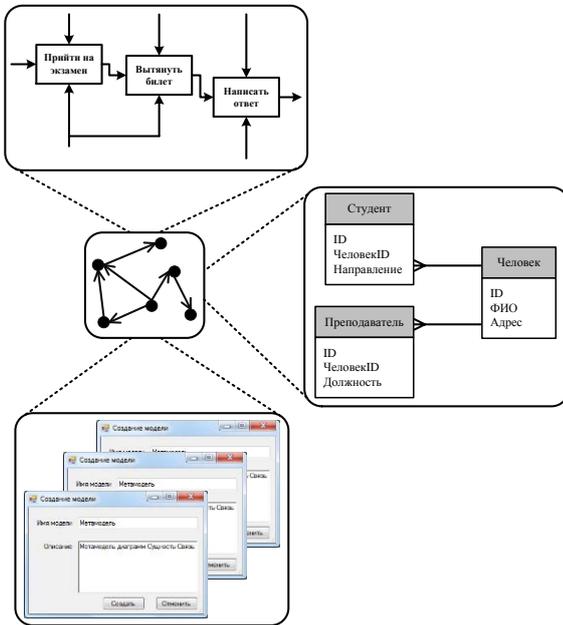


Fig. 1. Consideration of system objects from different points of view

Thus the software development process includes various types of activity in which different categories of users participate. It leads to creation and usage in systems designing of *various models* fulfilled from the different points of view, with different levels of details, which use for the description different modeling languages (see fig. 2).

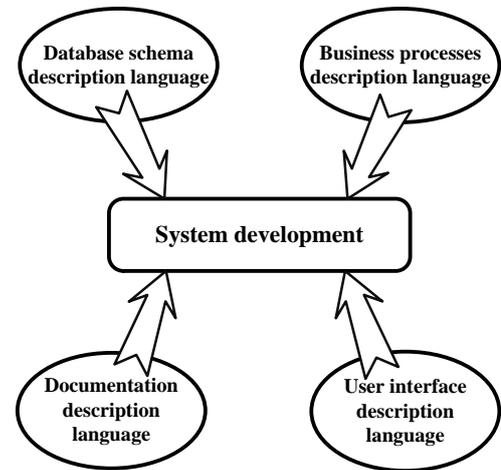


Fig. 2. The usage of different languages for software development

So there is a necessity of models transformation as between different levels of hierarchy, and within the same level between different modeling languages, for creation of united model of system and exporting of models to external systems (see fig. 3).

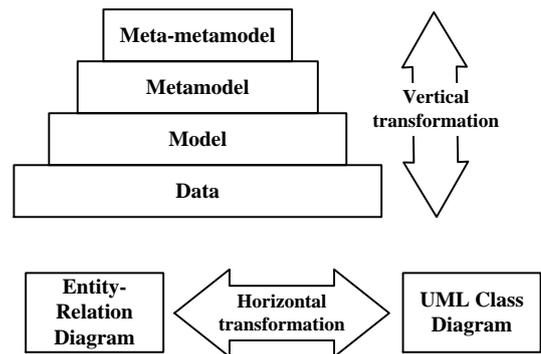


Fig. 3. Vertical and horizontal model transformations

In addition, there is an unresolved problem of models exporting from one information system to another (for example, business processes described in one system can not be executed in another, that these systems use various notations for business processes description).

Usage of domain-specific languages and tools for their creation also affects a transformation problem as there is a need of export of the created by the user models to external systems which, as a rule, use one of the standard modeling languages that is different from developed DSL. That is why one of the main components of the *MetaLanguage* system is the *Transformer*. This component uses graph grammars for transformations describing. Implementation of graph grammars in the *MetaLanguage* system is defined by assignment of this language workbench.

II. BASIC CONCEPTS

The basic concept of transformation definition is a *production rule* which looks like $p : L \rightarrow R$, where p is a *rule name*, L is a *left-hand side* of the rule, also called the *pattern*, and R is a *right-hand side* of the rule, which is called the

replacement graph. Rules are applied to the starting graph named the *host-graph*.

Let's suppose that four labeled graphs G, H, L, R are given, and graph L is a subgraph of graph G . Applying of the rule $p: L \rightarrow R$ to the starting graph G is called the replacement in graph G of subgraph L on graph R , which is a subgraph of graph H . The graph H is the result of this replacement.

Graph grammar is a pair $GG = (P, G_0)$, where P is a set of production rules, G_0 is a grammar starting graph.

Graph transformation is a sequenced applying to the starting labeled graph G_0 of finite set of rules

$$P = (p_1, p_2 \dots p_n) : G_0 \xRightarrow{p_1} G_1 \xRightarrow{p_2} \dots \xRightarrow{p_n} G_n.$$

List of production rules is arranged according to the chosen discipline, for example, by priorities. The transformation process is completed when the list of rules does not contain any rule which can be applied. There are also other disciplines of rules ordering, so some systems use the control mechanism to explicitly specify which rule should be applied as follows [5].

At transformations direction they can be classified as vertical and horizontal. *Vertical transformations* convert the models which belong to various hierarchy levels, for example, at mapping of the metamodel objects to domain model objects. *Horizontal transformation* is the conversion, in which the source and target models belong to one hierarchy level. An example of a horizontal transformation is a conversion of model description from one notation to another (see fig. 3).

The models are described with some modeling languages. Depending on the language on which source and target models are described, horizontal transformations can be divided into two types: endogenous and exogenous. *Endogenous transformation* is the transformation of the models which are described on the same modeling language. *Exogenous transformation* is the transformation of models which are described on various modeling languages [6].

Graph grammar often used to describe of any transformations, performed on graphs: definition of the models operational semantics [7, 8], the analysis of program systems with dynamic evolving structures [9, 10], etc. These grammars allow to describe the transformations that should occur in system at performance over it of the operations, specified in grammar.

The right-hand side of the rule may be not only a labeled graph, but the code on any programming language, and also a fragment of a visual model described in some notation. That is why the graph grammar can be used for generation syntactic correct models and for refactoring of existing models, code generation and model transformations from one modeling language to another [11].

Considering singularities and designation of MetaLanguage system, it is necessary to make the following requirements to its transformation component:

- To be obvious and easy to use for providing the

opportunity of involving to transformation description not only programmers, but also experts, specialists in domains. It can be achieved through the usage of visual notation of transformations description language.

- To allow using the created transformations directly in the system, i.e. to produce the models transformation in the same user interface in which they were designed.
- To perform both horizontal and vertical transformations, and availability of possibility to fulfill the horizontal transformations from one notation to another, including a "model-text" type.
- Metamodels from the left- and right-hand sides of the rule can be described by a user created metalanguage.
- To allow specifying the transformations of entities and relations attributes and constraints imposed on metamodel elements.

III. RELATED WORKS

There are various approaches to model transformations, some of them have the formal basis, so the systems AGG, GReAT, VIATRA use graph rewriting rules to perform transformations, and others apply technologies from other areas of software engineering, for example, the technique of programming by example.

Various modifications of the algebraic approach are implemented in systems AGG, GReAT, VIATRA. In AGG [12] the left- and right-hand sides of the production rule are the typed attribute graphs, both sides of a rule should be described in one notation, i.e. this system allows to fulfill only endogenous transformations that does impossible its usage in MetaLanguage system. Besides, this tool does not allow to make transformation of a "model-text" type. However the usage as the formal basis of the algebraic approach to graph transformations allows to produce graph parsing, to verify graph models, and the extension of graphs of Java possibilities makes transformations more powerful from a functional point of view.

The GReAT system [13] is based on the algebraic approach with double-pushout, therefore for transformation description it is necessary to create the domain that contains both the left- and right-hand sides of the production rule simultaneously with instructions of what element it is necessary to add, and what to remove. This form of rule is unusual for the end-user and a bit tangled. However it provides a possibility of execution the transformation of several source metamodels at once, which is significant advantage in comparison with other approaches. For metamodels definition the GReAT uses UML and OCL, it does not allow the user to choose the language of metamodels specification or to change its description. It makes this approach unsuitable for usage in MetaLanguage.

The QVT (Query/View/Transformation) is the proposed by OMG approach to models transformation, which provides the user with declarative and imperative languages [14]. Conversion is defined at the level of metamodels which is described on MOF. The advantage of this approach is the existence of standard of its description, and also usage of

standard languages OCL and MOF at the models transformation definition process. However these advantages also have the other side. Usage of MOF as a meta-modeling language, does not allow the user to choose a metalanguage convenient for him, or to change description of the metalanguage which is integrated in the QVT. In addition, this approach does not allow to make the transformation of a “model-text” type, since each metamodel should be described using the MOF standard. It imposes of some restrictions on a possibility of QVT usage in the MetaLanguage system.

VIATRA [15] is a transformation language, based on rules and patterns, which combines two approaches into a single specification paradigm: the algebraic approach for models description and the abstract state machines intended for exposition of control flow. Thanks to constructions of state machines the developers significantly raised the semantics of standard languages of patterns definition and graph transformation. Besides, powerful metalanguage constructions allow to make multi-level modeling of domains.

One of shortcomings of the VIATRA is an inexpressive textual language of metamodels description. Although the developers of approach have criticized the MOF standard for the lack of a possibility of multi-level modeling, they still remain within limits of this paradigm at usage of visual language for metamodels definition. VIATRA is not intended for execution of horizontal model transformations. Its main purpose is a verification and validation of the constructed models by their transformation.

The ATL is the language, allowing to describe transformations of any source model to a specified target model [16]. Transformation is performed at the level of the metamodels. The heart of ATL is the standard language of constraints description OCL.

The disadvantage of this language is high requirements to the conversion developer. Since ATL in most cases uses only textual definition of transformation, then in addition to knowledge of source and target metamodels the developer needs to know language of transformation definition. Lack of navigation on the target model complicates the process of rules determination.

The ATL is a dialect of QVT language and therefore inherits all its shortcomings. One of the differences from the QVT is very strict restriction on created transformations: the left-hand side of the rule should contain only one element. It highly complicates the development, increasing an amount of rules in system. All it does impossible the usage of this approach in the MetaLanguage system.

MTBE approach [17] is quite non-standard and unusual. The main purpose of MTBE is automatic generation of transformation rules on a basis of an initial set of learning examples. However implementations of this approach do not guarantee that the generation of model transformation rules is correct and complete. Moreover, the generated transformation rules strongly depend on an initial set of learning examples. Current implementations of MTBE approach allow to fulfill only full equivalent mappings of attributes, disregarding the complex conversions.

In summary, it is possible to say that all considered systems have some disadvantages which restrict their applicability for transformations definitions in the MetaLanguage system. But the most appropriate and perspective, from the author’s point of view, is the algebraic approach [12] with a single-pushout under condition of inclusion in it of some modifications:

- The availability of multi-level description of metamodels in the rule left- and right-hand sides.
- The description of transformation rules should be made at one level of hierarchy, and their application – on another.
- The existence of a possibility of exogenous transformations description.
- The right-hand side of production rule can contain as exposition of visual model, and some text.
- The availability of the opportunity to transform attributes of metamodel elements and constraints imposed on them.

Description of vertical transformations in MetaLanguage system has been considered explicitly in [4], therefore we will pass to reviewing of horizontal model transformations.

IV. HORIZONTAL MODEL TRANSFORMATIONS IN METALANGUAGE SYSTEM

All horizontal transformations are described at level of metamodels that allows to specify conversions which can be applied to all models created on basis of this metamodels. For a transformation creation it is necessary to select a source and target metamodels and to define production rules that are describing conversion.

To define the rule it is necessary to select objects (entities and relations) in a source metamodel, to set constraints on pattern occurrence and to define the right-hand side of the rule. Depending on a type of transformation a right-hand side will be a text template for code generation, or a fragment of a target metamodel.

Transformation rules are applied according to their order. At first all occurrences of a first rule pattern will be found, for each of them the system will fulfill a rule right-hand side, then the system will pass to the second rule and will begin to execute it, etc.

Let's assume that the system has selected next production rule of transformation and trying to execute it. For implementation of rule application it is necessary to describe two algorithms: the algorithm of the pattern search in the source host-graph and the algorithm of execution of a right-hand side of a rule.

A. Algorithm of the Pattern Search in the Host-graph

There are various algorithms of search of subgraph isomorphic to the given pattern [18]: Ullmann algorithm, Schmidt and Druffel algorithm, Vento and Foggia algorithm, Nauty-algorithm, etc. These algorithms are the most elaborated and often used in practice.

However difference of the proposed approach from the classical task of graph matching is that in this case it is necessary to find a pattern in the metamodel graph, i.e. it is required to lead matching of graphs which belong to various hierarchy levels, thus it is necessary to consider type of nodes and arcs, as between two nodes of the metamodel graph the several arcs of various type can be led [19].

The described algorithm for finding a pattern in the graph model is a kind of backtracking algorithm that takes exponential time.

Since the amount of arcs in the model graph is less than amount of the nodes usually, each arc uniquely identifies nodes, that are incident to it, and the degree of node can be more than two, that does not allow to select the following node of the model graph, entering into a pattern, it was decided to start subgraph search in a model graph on the basis of search of particular type arcs.

At the first step of algorithm all instances of some arbitrary relation of the pattern will be found, i.e. search of an initial arc with which execution of the second step of algorithm will begin is carried out. At the second stage it is necessary to find one of possible occurrence of all relations instances of the pattern-graph G_p in the source model graph G_S . At the third step necessary nodes will be add to target graph G_T and right-hand side of the rule will be execute.

The first step is a procedure *FindPattern*:

Algorithm 1. Procedure *FindPattern*

- 1.1. To clear the set of the source graph nodes viewed during search – *VisitedEntities*.
 - 1.2. To select from the pattern-graph G_p one of relations, denote it as *rel*. If there are not such relations, then go to the procedure *AddNodes* of adding of nodes in the graph G_T .
 - 1.3. To find all instances of the relation *rel* in the source model graph G_S . The set of these instances denote as *FoundRelations*.
 - 1.4. For each instance of the relation from the set *FoundRelations* execute procedure *FindSubGraph* to find a subgraph G_T , which corresponds to a pattern and contain the instance of relation *rel*, in the source model graph G_S .
-

Procedure of search of a subgraph containing the specified instance of relation *FindSubGraph* consists of following steps:

Algorithm 2. Procedure *FindSubGraph*

- 2.1. To add arc *rel* to the set of arcs of the required graph G_T .
 - 2.2. If after adding of arc it has appeared that the amount of arcs of the graph G_T equal the amount of arcs of the pattern-graph G_p then it is necessary to execute the procedure of nodes adding in the graph G_T , and then to return and remove arc *rel* from the set of arcs of the graph G_T , since in the source graph can exist other instances of the same type relation. Otherwise, go to step 2.3.
 - 2.3. To review the first node $entI_1$ which is incident to the arc
-

rel, if it does not belong to the set *VisitedEntities*:

- a. To add the node $entI_1$ to the set *VisitedEntities*.
 - b. To review all arcs of the graph G_S incoming to node $entI_1$, if the preimage some of them $fr^{-1}(rI_i^I)$ belongs to the pattern G_p and it was not considered earlier, it is necessary to search a subgraph, that contains an instance of the relation rI_i^I , starting from the second step of this algorithm.
 - c. To review all arcs of the graph G_S outgoing from node $entI_1$, if the preimage some of them $fr^{-1}(rI_i^O)$ belongs to the pattern G_p and it was not considered earlier, it is necessary to search a subgraph, that contains an instance of the relation rI_i^O , starting from the second step of this algorithm.
- 2.4. To consider the second node $entI_2$ which is incident to the arc *rel*, if it does not belong to the set *VisitedEntities*. Reviewing is made similarly to how it has been described in step 2.3.
 - 2.5. To execute the procedure of nodes adding to the graph G_T .
-

The procedure *AddNodes* of nodes adding to graph consists of three steps:

Algorithm 3. Procedure *AddNodes*

- 3.1. To consider all arcs of the graph G_T . If preimage of any node, that is incident to current arc, belongs to the pattern-graph G_p , it should be added to the set of nodes of the graph G_T .
 - 3.2. To find in the graph G_S nodes, preimages of which in the graph G_p are isolated, and add them in the graph G_T .
 - 3.3. To call the procedure of the rule right-hand side execution *ExecuteRightSide*. It is determined by a type of the transformation rule.
-

B. Algorithms of Rule Right-hand Side Execution

It is necessary to execute a right-hand side of production rule after the left-hand side subgraph has been found in a source graph. The algorithm of execution will depend on a type of transformation: whether transformation is a “model-model” or a “model-text”.

Transformation “model-text”. The transformation of this type allows the user to generate the source code on any target programming language on the basis of the constructed models as well as any other text representation of model, for example, its description on XML. In this case the right-hand side of production rule contains some template consisting of as static elements, which are independent of the found pattern, and dynamic parts, i.e. elements which vary depending on the found fragment of model.

For transformation fulfillment it is necessary to find all occurrences of a pattern in a source graph and to produce an insertion of an appropriate text fragment with a replacement of

a dynamic part by appropriate names of entities, relations, values of their attributes, etc.

The template is described in the target language. For selection of a dynamic part of a template the special metasympols are used: symbol “<<” (double opening angle brackets) to indicate the beginning of a dynamic part, “>>” (double closing angle brackets) to indicate the end of a dynamic part. As entities and relations can have the same name, then for entity describing before its name the prefix “E.” is specified, and for relation describing before its name the prefix “R.” is specified.

At the transformation specifying it is possible to set constraints on pattern occurrence. These constraints allow to define the context of the rule. They contain conditions with which found fragment of model should satisfy.

Let’s consider an example: define the transformation that allows on the basis of Entity-Relation Diagrams (ERD) to generate a SQL-query, building the schema of a corresponding database.

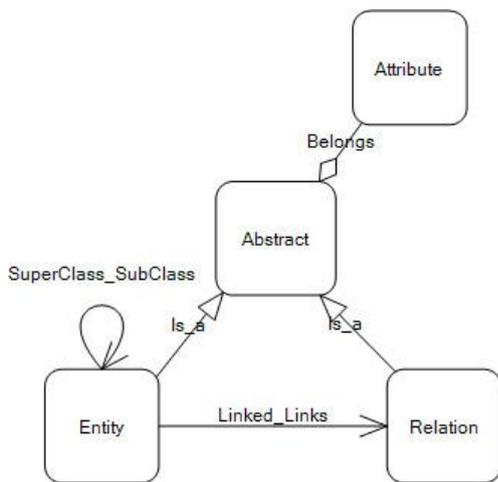


Fig. 4. Fragment of metamodel for Entity-Relation Diagrams

At the first step it is necessary to choose the metamodel of Entity-Relation Diagrams (see fig. 4) and to set the transformation rules. The metamodel contains the entities “Abstract”, “Attribute”, “Entity”, “Relation”. Attributes of the entity “Abstract” are “Name” that identifies an entity instance, and “Description”, containing the additional information about the entity. The entity “Abstract” is abstract, i.e. it is impossible to create instances of this entity in the model. “Abstract” acts as a parent for entities “Entity” and “Relation” (in the figure it is shown by an arrow with a triangular end). Both child entities inherit all parent attributes, relations, constraints. “Entity” does not have own attributes and constraints. “Relation” has the own attribute “Multiplicity”. The entity “Attribute” has following attributes: “Name”, “Type” and “Description”.

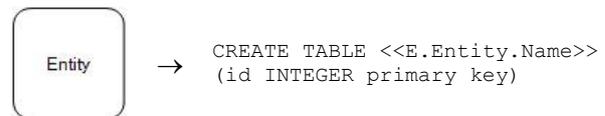
The bidirectional association “Linked_Links” connects entities “Relation” and “Entity”. It means that it is possible to draw equivalent relation between these entity instances in ERD-models. The second unidirectional association “SuperClass_SubClass” binds entity “Entity” with itself, it allows any instance of “Entity” to have parent (another instance

of “Entity”) in ERD-models. In ERD metamodel between entities “Attribute” and “Abstract” the aggregation “Belongs” is set (in figure this relation is represented by an arc with a diamond end), therefore in ERD-models instances of entities “Relation” and “Entity” can be connected by aggregation with the instances of entity “Attribute”.

For correct transformation execution the additional attributes in the source metamodel should be added. To determine what entity is a parent, and what entity is a child it is necessary to add the mandatory attributes of a reference type “Child” and “Parent” to relation “SuperClass_SubClass”. The entity “Relation” should be transformed to the reference between relational tables, therefore we will add to “Relation” additional mandatory attributes-references of “LeftEntity” and “RightEntity” and attribute of logical type “Has_Attribute”, which will facilitate the execution of the right-hand side of production rule.

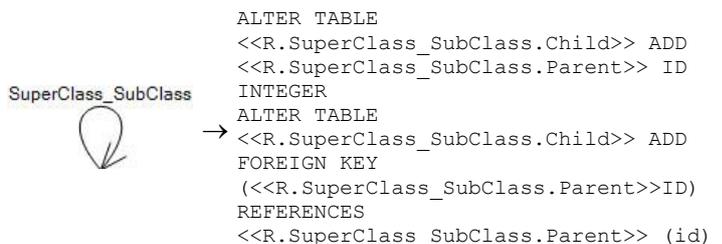
For transformation definition we will use the traditional rules of conversion of the ERD notation to a relational model, for this purpose we will define the following rules.

The rule “Entity” which transforms the instance of entity “Entity” to the single table looks like:

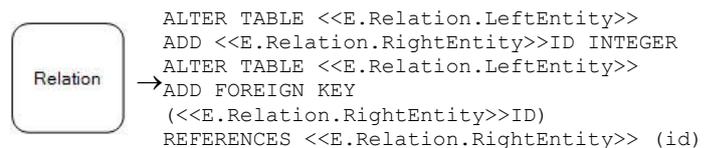


Here <<E.Entity.Name>> is a dynamic part of the template which allows to get a name of corresponding model entity.

As there is not inheritance relation in a relational model, it is necessary to specify the rule “Inheritance”, which for each instance of the relation “SuperClass_SubClass” in the “SubClass” table creates foreign key for connection with the “SuperClass” table. This rule looks like:



The rule “Relation_1M” allows to transform instance of entity “Relation”, which does not have attributes and its multiplicity is “1:M”, to the reference between tables. The rule has the following appearance:



In this rule at first in the table corresponding to the left entity the additional column with the name <<E.Relation.RightEntity>>ID is added, and then the foreign key (correspondence between this additional column

and a column containing the identifiers of right table rows) is created. This rule contains the constraint on pattern occurrence:

```
E.Relation.Multiplicity = 1:M AND
E.Relation.Has_Attribute = False
```

The rule “Relation_M1” allows to transform instance of entity “Relation”, which does not have attributes and its multiplicity is “M:1”, to the reference between tables. The rule looks like:

```
ALTER TABLE <<E.Relation.RightEntity>>
ADD <<E.Relation.LeftEntity>>ID INTEGER
ALTER TABLE
<<E.Связь.Relation.RightEntity>> ADD
FOREIGN KEY (<<E.Relation.LeftEntity>>ID)
REFERENCES <<E.Relation.LeftEntity>>(id)
```

The content of this rule right-hand side is similar to the content of the right-hand side of the rule “Relation_1M”. This rule contains the constraint on pattern occurrence:

```
E.Relation.Multiplicity = M:1 AND
E.Relation.Has_Attribute = False
```

For each instance of entity “Relation”, which has the attributes, or has the multiplicity “1:1” or “M:M”, it is necessary to create the single table that contains the key columns of each entity involved in relation. We call this rule “Relation_MM”, it has the following appearance:

```
CREATE TABLE <<E.Relation.Name>>
(id INTEGER primary key,
<<E.Relation.LeftEntity>>ID INTEGER,
<<E.Relation.RightEntity>>ID INTEGER)
ALTER TABLE <<E.Relation.Name>> ADD
FOREIGN KEY (<<E.Relation.LeftEntity>>ID)
REFERENCES <<E.Relation.LeftEntity>>(id)
ALTER TABLE <<E.Relation.Name>> ADD
FOREIGN KEY (<<E.Relation.RightEntity>>ID)
REFERENCES <<E.Relation.RightEntity>>(id)
```

This rule contains the constraint on pattern occurrence:

```
E.Relation.Multiplicity = M:M OR
E.Relation.Multiplicity = 1:1 OR
E.Relation.Has_Attribute = True
```

The rule “Attribute” adds the columns corresponding to attributes of instances of entities and relations to the created tables:

```
ALTER TABLE
<<E.Abstract.Name>> ADD
<<E.Attribute.Name>>
<<E.Attribute.Type>>
```

Let's consider an example, apply the described transformation to the model “University” presented in fig. 5.

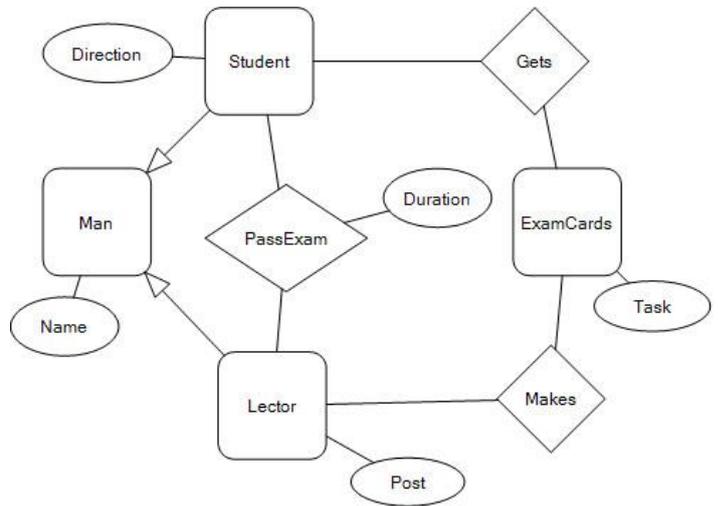


Fig. 5. Model “University” on the ERD notation

As a result the following text had been generated by the MetaLanguage system:

```
CREATE TABLE Man (id INTEGER primary key)
CREATE TABLE Student (id INTEGER primary key)
CREATE TABLE Lector (id INTEGER primary key)
CREATE TABLE ExamCards (id INTEGER primary key)
ALTER TABLE Lector ADD ExamCardsID INTEGER
ALTER TABLE Lector ADD FOREIGN KEY (ExamCardsID)
REFERENCES ExamCards (id)
ALTER TABLE ExamCards ADD StudentID INTEGER
ALTER TABLE ExamCards ADD FOREIGN KEY (StudentID)
REFERENCES Student (id)
CREATE TABLE PassExam (id INTEGER primary key,
StudentID INTEGER, LectorID INTEGER)
ALTER TABLE PassExam ADD FOREIGN KEY (StudentID)
REFERENCES Student (id)
ALTER TABLE PassExam ADD FOREIGN KEY (LectorID)
REFERENCES Lector (id)
ALTER TABLE Student ADD ManID INTEGER
ALTER TABLE Student ADD FOREIGN KEY (ManID)
REFERENCES Man (id)
ALTER TABLE Lector ADD ManID INTEGER
ALTER TABLE Lector ADD FOREIGN KEY (ManID)
REFERENCES Man (id)
ALTER TABLE Man ADD Name nvarchar(MAX)
ALTER TABLE PassExam ADD Duration nvarchar(50)
ALTER TABLE Lector ADD Post nvarchar(50)
ALTER TABLE Student ADD Direction nvarchar(MAX)
```

It should be noted that this transformation does not take into account complex conversions the ERD notation to the database schema, for example, those which would allow to create single dictionary table on the base of attribute, because it requires a special description language of templates and it is one of the areas for further research. Although such a conversion could be done by adding to the entity “Attribute” the attribute “Is_a_Dictionary” of logical type and setting the constraints on pattern occurrence.

Transformation “model-model”. Transformation of this type allows to produce conversion of model from one notation to another or to perform any operations over model (creation of new elements, reduction, etc.). Such transformation will allow to export model to external systems, and to provide the ability to convert the domain-specific language that was created by the

user in one of most common modeling language, for example, UML, ERD, IDEFO, etc.

The left-hand side of a production rule of this type transformation is a pattern, which is some fragment of the source metamodel, and the right-hand side of the rule is a some fragment of the target metamodel. At the production rule definition also it is necessary to describe the rules for converting the attributes of entities and relations. The created model should not contain dangling pointers, therefore the process of the transformation executions begins with the creation of entity instances and only then instances of relations are created. If in the process of model building the dangling pointers are still found the system will delete them.

At transformation execution it is necessary to consider the following elementary conversions:

- conversion “entity \rightarrow entity”;
- conversion “relation \rightarrow relation”;
- conversion “entity \rightarrow relation”;
- conversion “relation \rightarrow entity”.

Let's suppose that in the source model the instances of entities and/or relations of pattern are already found.

For fulfillment of the conversion $ee: Ent_L \rightarrow Ent_R$ it is necessary to create in the new model the instance $EntI_R$ of the appropriate entity of a rule right-hand side and to perform the specified transformation rules of attributes. The created instance of entity will have the same name, as the name of source entity instance.

For execution the conversion $rr: Rel_L \rightarrow Rel_R$ at first it is necessary to found in the source model the instances of entities $RelI_L.SEI$ and $RelI_L.TEI$, which are connected by the relation instance $RelI_L$, then the images of these instances $fe(RelI_L.SEI)$, $fe(RelI_L.TEI)$ should be found in the new model, and an instance of the relation from a rule right-hand side should be lead between them. After that it is necessary to fulfill transformation rules of attributes.

For fulfillment of the conversion $er: Ent_L \rightarrow Rel_R$ it is necessary to find in source model the nodes $EntI_S$, $EntI_T$ which are adjacent to entity instance $EntI_L$. Let's denote their images in the target model as $Source$ and $Target$. In the target model the relation instance $RelI_R$ between nodes $Source$, $Target$ should be lead. Further it is necessary to execute defined transformation rules of attributes. The algorithm of conversion $er: Ent_L \rightarrow Rel_R$ on the pseudocode can be described as follows:

Algorithm 4. Conversion “entity \rightarrow relation”

$EntI_L \leftarrow \text{Find_instance}(Ent_L, G_S);$
 $EntI_S \leftarrow \text{Find_adjacent_node}(EntI_L);$
 $EntI_T \leftarrow \text{Find_adjacent_node}(EntI_L);$

$Source \leftarrow \text{Find_node_image}(EntI_S);$
 $Target \leftarrow \text{Find_node_image}(EntI_T);$
 $RelI_R \leftarrow \text{Add_new_arc}(Source, Target);$
 $\text{Execute_attributes_transformation}(EntI_L, RelI_R);$

The complexity of the function “Find_instance” is equal to $O(N)$, where N is an amount of instances of various entities in model. The complexity of the function “Find_adjacent_node” is equal to a constant, since for its performance it is necessary to pass on the corresponding arc of the graph model. To find the image of node it is necessary to pass on arc-reference, i.e. the complexity of function “Find_node_image” is equal to a constant. The complexity of executing of function “Execute_attributes_transformation” is equal to $O(\sum_{i=1}^k A_i)$, where k is an amount of specified transformation rules of attributes, A_i is the complexity of the performance of i -th rule. Thus, the complexity of the presented algorithm is equal to $O(\sum_{i=1}^k A_i + N)$.

Conversion $re: Rel_L \rightarrow Ent_R$ transforms the instance of relation $RelI_L$ found in the source model to the entity instance $EntI_R$ of target model. For conversion execution it is necessary to create the entity instance $EntI_R$, to perform the specified transformation rules of attributes. The name of $EntI_R$ will be the same as the name of the relation instance $RelI_L$. At the next step it is necessary to find entities instances $RelI_L.SEI$, $RelI_L.TEI$, which are connected by relation instance $RelI_L$.

Further the instances of relations that connect an entity instance $EntI_R$ with nodes $Source$ and $Target$, which are images of the nodes $RelI_L.SEI$ and $RelI_L.TEI$, accordingly, with keeping of orientation of relation instance.

Thus, the conversion algorithm will be following:

Algorithm 5. Conversion “relation \rightarrow entity”

$RelI_L \leftarrow \text{Find_instance}(Rel_L, G_S);$
 $\text{Add_new_node}(EntI_R, G_T.V);$
 $\text{Execute_attributes_transformation}(RelI_L, EntI_R);$
 $Source \leftarrow \text{Find_node_image}(RelI_L.SEI);$
 $Target \leftarrow \text{Find_node_image}(RelI_L.TEI);$
 $\text{Add_new_arc}(Source, EntI_R);$
 $\text{Add_new_arc}(EntI_R, Target);$

The complexity of algorithm of conversion “relation \rightarrow entity” performance is equal to $O(\sum_{i=1}^k A_i + N)$.

It is possible to present the rest conversions of “model-model” type by a combination of these elementary operations.

Let's consider an example, perform the transformation of the model on Entity-Relation Diagrams notation to UML Class Diagrams.

Since the transformation is done at the metamodel level, then at the first step it is necessary to create/open source and target metamodels. The ERD metamodel was presented in the fig. 4. Metamodel of UML Class Diagrams is shown in the fig. 6. It contains the following elements: the entity "Class" and three relations "Inheritance", "Association", "Aggregation". Let's define the production rules that determine the transformation.

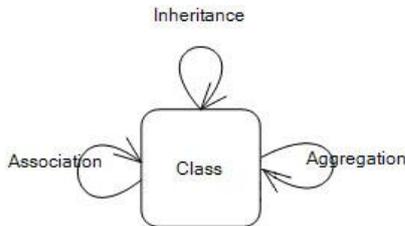
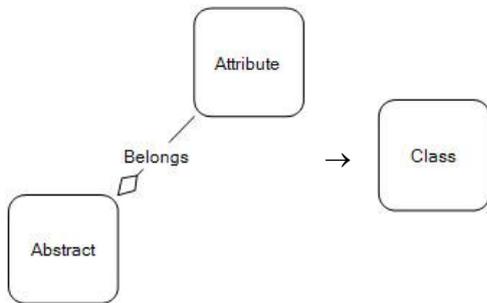
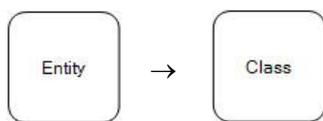


Fig. 6. Fragment of metamodel for Class Diagrams

The rule "Abstract-Class" allows to convert the instances of entities "Entity" and "Relation", which are connected at least with one instance of entity "Attribute", to the instance of entity "Class". This rule has the following appearance:

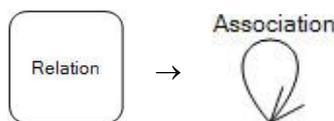


The rule "Entity-Class" allows to convert the instance of entity "Entity", which is not associated with any instance of the entity "Attribute", to the instance of an entity "Class". The rule has the following form:



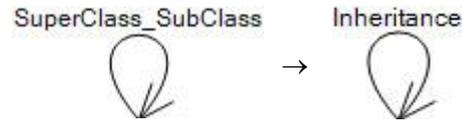
The rule "Relation-Association" converts instances of the entity "Relation" of the source model to instances of the relation "Association" of the target model.

This rule looks like:



The rule "Inheritance" puts in correspondence to each instance of the relation "SuperClass_SubClass" of source

model a particular instance of the relation "Inheritance" of target model. This rule has the following form:



After definition of all rules, which are included in the transformation, it is possible to execute conversion on a specific model. Let's perform this transformation on the considered earlier model "University" (see fig. 5). The result of the transformation execution is presented in fig. 7.

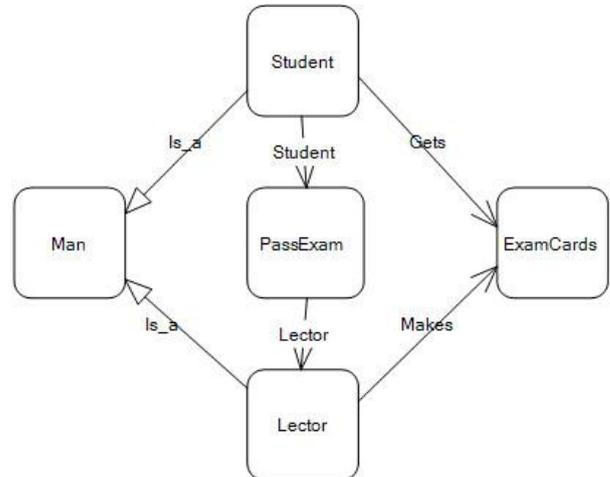


Fig. 7. Model "University" on the Class Diagrams notation, generated by MetaLanguage system

V. CONCLUSION

Models transformations are a central part of the model-based approach to system development, since an existence in one system of models fulfilled from the different points of view, with a different level of detail and using for the description different modeling languages, demands presence of model transformation tools both between various levels of hierarchy, and within single level: at transition from one modeling language to another.

The presented approaches have been implemented in a transformer of MetaLanguage system. This component allows to convert models, described on visual domain-specific languages, to text or other graphic models. The component has a convenient and simple user interface, therefore not only professional developers, but also domain specialists, for example, business analysts, can work with it.

REFERENCES

- [1] Брыксин Т.А., Литвинов Ю.В. Среда визуального программирования роботов QReal:Robots / Материалы международной конференции "Информационные технологии в образовании и науке". Самара, 2011. – С. 332-334.
- [2] Демаков А. Язык описания абстрактного синтаксиса TreeDL и его использование / Препринт ИСП РАН. – 2006. – № 17. – С. 1-24.
- [3] Межуев В.И. Предметно-ориентированное моделирование распределенных приложений реального времени / Системы обработки информации. – 2010. – № 5(86). – С. 98-103.

- [4] 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 (SYRCoSE 2012). М.: Изд-во Института системного программирования РАН, 2012. – P. 42-53.
- [5] Сухов А.О., Серый А.П. Использование графовых грамматик для трансформации моделей / Материалы конференции "CSEDays 2012". Екатеринбург: Изд-во Урал. ун-та, 2012. – С. 48-55.
- [6] Mens T., Czarnecki K., Gorp P.V. A Taxonomy of Model Transformations / Electronic Notes in Theoretical Computer Science. Amsterdam: Elsevier Science Publishers, 2006. Vol. 152. – P. 125-142.
- [7] Подкопаев А.В., Брыксин Т.А. Генерация кода на основе графической модели / Материалы межвузовского конкурса-конференции студентов, аспирантов и молодых ученых Северо-Запада "Технологии Microsoft в теории и практике программирования". СПб.: Изд-во СПбГПУ, 2011. – С. 112-113.
- [8] Montanari U., Rossi F. Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems // Applied Categorical Structures. Netherlands: Springer, 1999. – P. 333-370.
- [9] Миков А.И., Борисов А.Н. Графовые грамматики в автономном мобильном компьютеринге / Математика программных систем: межвуз. сб. науч. ст. / под ред. А.И. Микова, Л.Н. Лядовой. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 50-59.
- [10] König B. Analysis and Verification of Systems with Dynamically Evolving Structure / Habilitation thesis. – 238 p. [Электронный ресурс]. URL: <http://jordan.inf.uni-due.de/publications/koenig/habilschrift.pdf> (дата обращения: 17.02.2013).
- [11] Rekers J., Schuerr A. A Graph Grammar approach to Graphical Parsing / Proceedings of the 11th IEEE International Symposium on. Washington: IEEE Computer Society, 1995. – P. 195-202.
- [12] Ehrig H., Ehrig K., Prange U. et al. Fundamentals of Algebraic Graph Transformation. New York: Springer-Verlag, 2006. – 388 p.
- [13] Balasubramanian D., Narayanan A., Buskirk C.P. et al. The Graph Rewriting and Transformation Language: GReAT / Electronic Communications of the EASST. – 2006. – Vol. 1. – P. 1-8.
- [14] Gardner T., Griffin C., Koehler J. et al. A review of OMG MOF 2.0 Query/Views/Transformations Submissions and Recommendations towards the final Standard / Proceedings of the 1st International Workshop on Metamodeling for MDA. York, 2003. – P. 1-20.
- [15] Csertan G., Huszerl G., Majzik I. et al. VIATRA – Visual Automated Transformations for Formal Verification and Validation of UML Models. Available at: http://static.inf.mit.bme.hu/pub/ase2002_varro.pdf (accessed 17 March 2013).
- [16] ATL: Atlas Transformation Language / ATL Starter's Guide. LINA & INRIA Nantes, 2005. – 23 p.
- [17] Wimmer M., Strommer M., Kargl H. et al. Towards Model Transformation Generation By-Example / Proceedings of the 40th Annual Hawaii International Conference on System Sciences. Washington: IEEE Computer Society, 2007. – P. 1-10.
- [18] Серый А.П. Алгоритмы сопоставления графов для решения задач трансформации моделей на основе графовых грамматик / Математика программных систем: межвуз. сб. науч. ст. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 60-73.
- [19] Лядова Л.Н., Серый А.П., Сухов А.О. Подходы к описанию вертикальных и горизонтальных трансформаций метамodelей / Математика программных систем: межвуз. сб. науч. ст. Пермь: Изд-во Перм. гос. нац. исслед. ун-та, 2012. – Вып. 9. – С. 33-49.

An Approach to Graph Matching in the Component of Model Transformations

Alexander P. Seriy
Department of Software and Computing
Systems Mathematical Support
Perm State University
Perm, Russian Federation
E-mail: SerAlexandr@bk.ru

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 – Nowadays approaches, based on models, are used in the development of the information systems. The models can be changed during the system development process by developers. They can be transformed automatically: visual model can be translated into program code; transformation from one modeling language to other can be done. The most appropriate way of the formal visual model presentation is metagraph. The best way to describe changes of visual models is the approach, based on graph grammars (graph rewriting). It is the most demonstrative way to present the transformation. But applying the graph grammar to the graph of model means to find the subgraph isomorphic to the left part of the grammar rule. This is an NP-complete task. There are some algorithms, developed for solving this task. They were designed for ordinary graphs and hypergraphs. In this article we consider some of them in case of using with the metagraphs representing models.

Keywords – subgraph isomorphism, metagraphs, graph grammars, model transformations.

I. INTRODUCTION

Nowadays approaches, based on models, are used at information systems development (Model Driven Design, Model Driven Engineering, Model Based Development, etc.). A graph is the most obvious way to represent a visual model. As shown in [1], using domain-specific models is the most convenient way of representing information about the system.

The created models can be changed during the system development process by developers (data base designers, system analysts). The developed models can be transformed automatically: visual model can be translated into program code; transformation from one modeling language to other can be done. Therefore, the task of transformation rules development is important for information system developers.

There are some approaches to create a special language and automatically generate model transformation rules using this language. Thus, the Model Driven Architecture (MDA) [5] involves the construction of two domain models –

platform independent (PIM) and platform-specific models (PSM). In this case the platform-specific model can be constructed automatically.

The most appropriate way to describe the changes is an approach based on graph grammars. Graph grammars provide a powerful tool of describing transformation of models. However, in their work, these tools should solve the problem of finding a subgraph isomorphic to a given graph. This is a NP-complete problem. There are some efficient algorithms, developed for solving this problem, and many of them are applicable for model transformation by graph grammars. However, all of them were originally designed for digraphs or hypergraphs. As we are going to use the metagraphs, we should consider the applicability of the existing algorithms to metagraphs and evaluate the effectiveness of these algorithms in this case.

II. GRAPH MATCHING ALGORITHMS

Graph is an ordered pair $G = (V, E)$, where $V = \{v_1, \dots, v_n\}$ is a non-empty set of vertices of the graph and $E = \{e_1, \dots, e_m\}$ is the set of edges of the graph.

The graph in which we need to find and replace subgraph usually is called “host graph”, and the graph we need to find is called “sought-for graph”.

The most of theoretical research in graph theory was conducted specifically for ordinary graphs; in particular, there are some algorithms for comparing graphs. We will consider the following algorithms: the Ulman algorithm; the Schmidt-Druffel algorithm; the Vento and Foggia algorithm; Nauty-algorithm; the algorithm for checking the isomorphism of colored hypergraphs basing on easy-to-compute parameters of graph; the algorithm of checking the structure of the neighbors for directed hypergraphs and checking isomorphism by invariants.

Ullman algorithm. Ullmann algorithm [8] is one of the first algorithms proposed for solving the problem of graph isomorphism. It is a *backtracking algorithm*, but it uses the *refining procedure* (Listing 1) to reduce search field.

Algorithm constructs a subgraph, which is suspected to be isomorphic to the sought-for graph. At each step the algorithm tries to add to constructing subgraph a new vertex (V for the host-graph and v to the sought-for graph). After that, for each vertex v_1 of the sought-for graph adjacent to the vertex v ; function *Refine* is trying to find a vertex V_1 in the host graph, such as: V_1 is connected to V , and $deg(V_1) \geq deg(v_1)$. If a match is found, the function *Refine* returns “Ok” and constructed subgraph will be extended on the next step. Otherwise, the function returns failure and algorithm will fall back on the search tree.

LISTING 1. REFINE PROCEDURE

```
bool Refine(graph Host, graph Small)
{
  foreach (node n in Host.Nodes)
  {
    bool Found=false;
    foreach (node n1 in Small.Nodes)
    {
      if (n.Degree()>n1.Degree())
      {
        Found=true;
        break;
      }
    }
    if (!Found)
      return false;
  }
  return true;
}
```

The lower boundary of the time complexity of this algorithm is $O(N^3)$, the upper – $O(N^3 \times N!)$.

Schmidt-Druffel algorithm. Schmidt-Druffel algorithm [7] is a *backtracking algorithm*, which is using a *matrix of distances* between vertices of the graph to reduce the space of search. Using this matrix, the *characteristic matrix* of size $N \times (N-1)$ is built (Listing 2). The element c_{ij} of a characteristic matrix is the number of vertices in the graph, which are placed at the distance j from vertex i .

LISTING 2. BUILDING CHARACTERISTIC MATRIX

```
Matrix BuildCharMatrix(graph g)
{
  Matrix result = ClearCharacteristicMatrix
    (g.NodesCount, g.NodesCount-1);
  for (int n=0;n<NodesCount;n++)
  {
    for (j=0;j<NodesCount;j++)
    {
      int dist=g.GetDist(i, j);
      result [i][dist]++;
    }
  }
  Return result;
}
```

The vertices of the host-graph are divided into classes after constructing such a matrix (Listing 3). All vertices shall be in the same class, if their columns in the characteristic matrix are equal.

LISTING 3. BUILDING CLASSES OF VERTICES

```
List<List<int>> BuildClasses(Matrix CharMatrix)
{
  int i,j;
  List<List<int>> result=new List<List<int>>();
  result.Add(new List<int>());
  result[0].Add(0);
  for (i=1;i<CharMatrix.ColumnsCount;i++)
  {
    for (j=0;j<result.Count;j++)
    {
      if (CharMatrix.Columns[i] ==
          CharMatrix.Columns[result[j][0]])
      {
        result[j].Add(i);
        break;
      }
    }
    if (j==result.Count)
    {
      result.Add(new List<int>());
      result[result.Count-1].Add(i);
    }
  }
}
```

After that, the vertices of the sought-for graph should be attributed to the already existing class, so columns of the characteristic matrix of the sought-for graph compares with the columns of the characteristic matrix of the host graph.

Thus such a relationship is built between the vertices of the two graphs, which preserve the classes of vertices. As a result, the partition to the classes can reduce the dimension of the problem, at best, by reducing it to the trivial, when all classes have only one vertex. However, the partition can not be useful at all, if all vertices will be in the same class.

The lower boundary of the time complexity of this algorithm is $O(N^2)$, the upper – $O(N \times N!)$.

Vento and Foggia algorithm. Vento and Foggia’s *genetic algorithm* [9] is an algorithm developed for solving the problem of finding a subgraph isomorphic to a given graph. Starting with a set of subgraphs, algorithm calculates the fitness function for them, which characterizes their similarity to the original graph. After calculating of the fitness function, the new generation of the subgraphs is building. A set of easy-to-compute graph invariants is often taken as the fitness function. The functions listing below can be used as the *invariants*.

1. Ordered set of vertices degrees (Listing 4).

LISTING 4. EVALUATING THE INVARIANT
“SET OF VERTICES DEGREES”

```
List<int> GetDegrees(graph g)
{
  List<int> result=new List<int>();
  foreach (node n in g.Nodes)
  {
    result.Add(n.degree);
  }
  result.Sort();
  return result;
}
```

2. The characteristic path length – the average length of the shortest paths between each pair of vertices (Listing 5).

LISTING 5. EVALUATING THE INVARIANT
“CHARACTERISTIC PATH LENGTH”

```
double AverageDist(graph g)
{
    Double res=0;
    For (int i=0;i<g.NodesCount;i++)
    {
        For (int j=0;j<g.NodesCount;j++)
        {
            res+=g.GetDist(i,j);
        }
    }
    return res/n/n;
}
```

3. Number of second neighbors (the vertices adjacent to the neighbors of this one) for each vertex. The numbers are ordered ascending (Listing 6).

LISTING 6. EVALUATING THE INVARIANT
“NUMBER OF SECOND NEIGHBORS”

```
List<int> SecondNeighbors(graph g)
{
    List<int> result=new List<int>();
    int t;
    foreach (node n in g.Nodes)
    {
        t=0;
        foreach (node n1 in g.Nodes)
        {
            if(g.GetDist(n,n1)==2)
            {
                t++;
            }
        }
        result.Add(t);
    }
    result.Sort();
    return result;
}
```

4. The number of paths between the vertices x and y , passing through the vertex i .

Other functions can be used as the invariants too.

The boundaries of the algorithm depend on the selected set of invariants. Author’s fitness function gave the following boundaries: the lower boundary of the algorithm is $O(N^2)$, the upper – $O(N \times N!)$

The later modification of the algorithm [9], named VF2, exists. It has the same complexity boundaries, but smaller hidden constants. The authors of this algorithm have shown [10] that their algorithm is faster than the Schmidt-Dryuffel algorithm.

Nauty-algorithm. This algorithm is designed by B. McKay [4]. The Nauty-algorithm uses a tightening

transformation in order to bring graph to its canonical code. A code that is the same for isomorphic graphs and not the same for non-isomorphic is named canonical. After the construction of a canonical code the isomorphism checking becomes trivial task. The Nauty-algorithm is considered as the fastest algorithm known to nowadays.

The algorithm divides the set of vertices into classes basing of the special properties of the vertices.

B. McKay gave his implementation of the Nauty-algorithm in the public domain. In this implementation he uses a significant number of optimizations and means of reducing the search, such as “granted automorphisms”. The author admitted that not all of the optimization techniques used by him are documented.

III. HYPERGRAPH MATCHING ALGORITHMS

Hypergraph is a pair $G = (X, E)$, where X is a non-empty set of objects of a certain nature, called *vertices* of the hypergraph, and E – a family of non-empty subsets of X , named *hyperedges*.

Algorithm for checking the isomorphisms of colored hypergraphs basing on easy-to-compute parameters of graph [2]. This algorithm is a *combination of the “divide and conquer” approach and dynamic programming*. First, the vertices of both graphs are divided into classes (*Cosets*) in order to reduce the problem of graph isomorphism to problems in the theory of permutation groups, in particular, to the problem of intersection classes. Then these problems can be solved by dynamic programming.

The computational complexity of the algorithm – $2^{O(b)} \times N^{O(1)}$, where b – the maximum number of nodes of the same color.

Algorithm of checking the structure of the neighbors for directed hypergraphs [3]. It is an *improved backtracking algorithm*. Before adding a new vertex V in the expanding subgraph, this algorithm counts the number of different paths of a certain length for each vertex, which is connected to V . Resulting set of numbers is named a structure of the adding vertex. With such information algorithm checks whether it is possible to expand the subgraph further, and if not, algorithm will fall back.

The authors of the algorithm do not lead to count of the complexity. However they compare this algorithm [3] with the algorithm VF2 (Vento-Foggia 2). While checking algorithm structure neighbors greatly reduced number of analyzed variants, each test takes too much time. As a result, the algorithm is almost always slower than the algorithm VF2.

Checking isomorphism by invariants. This algorithm [6] involves the invariants to compare hypergraphs. The authors propose an algorithm to consider a number of invariants to more quickly identify nonisomorphic graphs. These invariants can be, for example, a set of ordered vertex degrees, the lowest path length between each pair of vertices, the number of entries in each of the graphs of the same

subgraphs of smaller dimensions (e.g., the number of cycles of length 3), etc.

This algorithm is developed to solve the problem of testing isomorphism of two graphs. However, it can be applied to the problem of finding isomorphic subgraph. Such invariants as the length of the shortest paths between vertices and vertex degrees will no longer be useful in this case, but the invariant “number of entries in each of the graphs of the same subgraphs smaller” can be adapted to subgraph search.

The problem of this approach is that it can determine only the difference of graphs. If all the above graphs matched invariants coincide, this does not guarantee isomorphism. The authors propose to increase the number of invariants to increase the likelihood of a negative response to the issuance of non-isomorphic graphs.

IV. METAGRAPH MATCHING ALGORITHMS

Metagraph is an ordered pair $G = (X, E)$, where $X = \{x_i\}$ ($i = \overline{1, n}$) is a finite nonempty set of metaverices, E – the set of edges of the graph. Each edge $e_k = (V_i, W_i)$, $k = \overline{1, m}$, $V_i, W_i \subseteq X$ and $V_i \cup W_i \neq \emptyset$, that is, each edge in metagraph connects two subsets of vertices.

It is shown [1] that the most convenient way to represent the domain model is metagraph. This leads us to the problem of searching metasubgraph isomorphic to the given one in order to execute graph rewriting at model transformation process.

In this section we will consider the applicability of existing graph matching algorithms to metagraphs.

Ullman algorithm. The most flexible element of the algorithm is a function *Refine*. We can make it to check not only the degree of vertices, but the number of subvertices in the metavertex (Listing 7).

LISTING 7. FUNCTION REFINE, MODIFIED FOR METAGRAPH

```
bool Refine(graph Host, graph Small)
{
    foreach (node n in Host.Nodes)
    {
        bool Found=false;
        foreach (node n1 in Small.Nodes)
        {
            if (n.Degree()>n1.Degree()
                && n.SubNodes.Count ==
                n1.SubNodes.Count)
            {
                Found=true;
                break;
            }
        }
        if (!Found) return false;
    }
    return true;
}
```

Although we can quicker weed out unsuitable subgraphs, it does not affect the evaluation of the algorithm, but only reduces the hidden constants.

The lower boundary of the algorithm complexity is $O(N^3)$, the upper – $O(N^3 \times N!)$.

Schmidt-Druffel algorithm. This algorithm can be optimized as follows: in the division into classes of vertices we may consider not only the value of the characteristic matrix, but the number of subvertices (Listing 8).

LISTING 8. MODIFIED CONDITION OF VERTICES PARTITION

```
if (CharMatrix.Columns[i] ==
    CharMatrix.Columns[result[j][0]]
    && g.Nodes[i].SubNodes.Count ==
    g.Nodes[result[j][0]].SubNodes.Count)
{
    result[j].Add(i);
    break;
}
```

So we will get more classes, and reduce the likelihood of a worse situation when all the vertices are included in the same class. Thus, the estimates will not change, but the distribution of probabilities of the worst and the best situation will improve.

The lower boundary of the algorithm complexity is $O(N^2)$, the upper – $O(N \times N!)$.

Vento and Foggia algorithm. Efficiency of this algorithm depends on the used set of invariants. The most obvious invariant for metagraphs is an ordered set of capacities of metaverices (Listing 9).

LISTING 9. EVALUATING THE INVARIANT
“ORDERED SET OF CAPACITIES OF METAVERTICES”

```
List<int> SubNodesCounts(graph g)
{
    List<int> result=new List<int>();
    foreach (node n in g.Nodes)
    {
        result.Add(n.SubNodes.Count);
    }
    result.Sort();
    return result;
}
```

Adding such an invariant will decrease the hidden constant in the estimates of the complexity. The lower boundary of the algorithm complexity $O(N^2)$, the upper – $O(N \times N!)$.

Nauty-algorithm. Nauty-algorithm differs from the previously discussed algorithms. The process of constructing the canonical code is not changed for graphs and metagraphs. We can assume that the vertices belonging to the metavertex – a new special property of vertex. It allows us to perform the first step of the algorithm automatically. As this algorithm is the fastest, it is a main candidate for the implementation in the model transformation component of MetaLanguage system. The algorithm implementation suggested by B. McKay is useless for us – it takes just a data structure that stores the graph. This representation does not allow to transfer a set of vertices belonging to metavertex.

Algorithm for checking the isomorphisms of colored hypergraphs basing on easy-to-compute parameters of graph. When we try to apply this algorithm to metagraphs, the number of subvertices in metavertex will be considered a

special color of the vertex. If the original graph is colored, it grinds partition by color and reduces the number of vertices of each color. However, the complexity of this algorithm is always $2^{O(b)} \times N^{O(1)}$. It can be a significant disadvantage because the other algorithms can work faster in the average.

Algorithm of checking the structure of the neighbors for directed hypergraphs. This algorithm can not use the information of vertices in metaverter (only path lengths are important to this algorithm), and it is often slower than the algorithm Vento-Foggia (VF2). So this algorithm is useless in practice.

Checking isomorphism by invariants. This approach can be applied to regular graphs, and to hyper- and metagraphs, but it can say only that the sought-for subgraph is in the graph or not, but can not identify the vertices that form it. Thus this method is useless for solving the problem of graphs transformation with graph grammars. However, it can be used in conjunction with any other algorithm for the preliminary analysis. If the algorithm reports that there is no subgraph isomorphic to a given, running of a more powerful algorithm is not necessary.

As it seems from the Table 1, the leadings algorithms are the algorithm Vento and Foggia and Nauty algorithm.

TABLE 1. THE COMPARISON OF THE ALGORITHMS

Algorithm	Best-case complexity	Worth-case complexity	Always finds correct answer
Ullman algorithm	$O(N^2)$	$O(N \times N!)$	+
Schmidt-Druffel algorithm	$O(N^2)$	$O(N \times N!)$	+
Vento and Foggia algorithm	$O(N^2)$	$O(N \times N!)$	+
Nauty-algorithm	Not leaded by the author	Not leaded by the author	+
Algorithm for checking the isomorphism of colored hypergraphs basing on easy-to-compute parameters of graph	$2^{O(b)} \times N^{O(1)}$	$2^{O(b)} \times N^{O(1)}$	+
Algorithm of checking the structure of the neighbors for directed hypergraphs	Not leaded by the authors, more than for Vento and Foggia algorithm	Not leaded by the authors, more than for Vento and Foggia algorithm	+
Checking isomorphism by invariants	Depends on the chosen invariants	Depends on the chosen invariants	-

V. CONCLUSION

Graph matching is important task for implementation of DSM-platform, where new visual domain specific modeling languages (DSML) are created and model transformation rules based on graph grammars are defined.

This article covers seven graph matching algorithm in the case of their applicability to the metagraphs comparison in order to search subgraph of model metagraph. All of them can be applied to compare metagraphs, and many of them can use the features of the metagraphs structure to get some acceleration. However, the difference in the complexity is only a constant for all of them.

Our analysis revealed the two leaders – the algorithm Vento and Foggia and Nauty algorithm. We plan to implement both of them and test them to identify the most effective algorithm to execute graph matching in component of visual model transformation, included in MetaLanguage DSM-platform.

REFERENCES

- [1] Сухов А.О. Анализ формализмов описания визуальных языков моделирования // Современные проблемы науки и образования. – 2012. – № 2; URL: www.science-education.ru/102-5655 (дата обращения: 10.11.2012).
- [2] Arvind V., Das B., Köble J., Toda S. Colored Hypergraph Isomorphism is Fixed Parameter Tractable // In Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science, 2010.
- [3] Battiti R., Mascia F. An Algorithm Portfolio for the Sub-Graph Isomorphism Problem, Universit' a degli Studi di Trento.
- [4] McKay B.D. Practical Graph Isomorphism // Congressus Numerantium. – 1981. – №30. – с. 45-87.
- [5] MDA Guide Version 1.0. OMG document, Miller, J. and Mukerji, J. Eds., 2003. Available: <http://www.omg.org/docs/omg/03-06-01.pdf> [19.06.2012].
- [6] Remie V. Bachelors Project: Graph isomorphism problem, Eindhoven University of Technology Department of Industrial Applied Mathematics, 2003.
- [7] Schmidt D., Druffel L. A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices // Journal of the Association for Computing Machinery. – 1976. – №23. – P. 433-445.
- [8] Ullmann J.R. An Algorithm for Subgraph Isomorphism // Journal of the Association for Computing Machinery. – 1976. – №23. – P. 31-42.
- [9] Vento M., Foggia P., Sansone C. An Improved Algorithm for Matching Large Graphs // IAPR-TC-15 International Workshop on graphbased Representations. – 2001. – №3. – P. 193-212.
- [10] Vento M., Foggia P., Sansone C. A Performance Comparison of Five Algorithms for Graph Isomorphism // In Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition, 2001.

Technology Aspects of State Explosion Problem Resolving for Industrial Software Design

P.Drobintsev, V.Kotlyarov, I.Nikiforov

Saint-Petersburg State Polytechnic University

e-mail: vpk@spbstu.ru

Abstract – The paper describes technology aspects of proposed solution for resolving of state explosion problem. The main point is in usage of guides which can be created both manually and automatically and allow to reduce state space during trace generation process. The following techniques is described: traceability tracking of requirements, guides generation based on selected criterion, guides analysis in case of problems with traces generation.

Usage of described techniques in verification and testing phases of software development allow to resolve problem of state exposure for industrial projects.

Key words — design specification, state explosion problem, requirements, behavioral tree, guides, guide formalization.

I. INTRODUCTION

One of the main problems in development and testing automation of industrial applications' software is handling of complicated and large scale requirements specifications. Documents specifying requirements specifications are generally written in natural language and may contain hundreds and thousands of requirements. Thereby the task of requirements formalization to describe behavioral scenarios used for development of automatic tests or manual test procedures is characterized as a task of large complexity and laboriousness.

Applicability of formal methods in the industry is determined to a great extent by how adequate is the formalization language to accepted engineering practice which involves not only code developers and testers but also customers, project managers, marketing and other specialists. It is clear that no logic language is suitable for adequate formalization of requirements which would keep the semantics of the application under development and at the same time would satisfy all concerned people [1].

In modern project documentation the formulation of initial requirements is either constructive, when checking procedure or scenario of requirement coverage checking can be constructed from the text of this requirement in natural language, or unconstructive, when functionality described in the requirement does not contain any explanation of its checking method.

For example, behavioral requirements of telecommunication applications in case of described scenario of coverage are constructively specified and assume allow using of verification and testing for realization checking. Non-behavioral requirements are usually unconstructively specified and require additional information during formalization which allows reconstructing the checking scenario, i.e. converting of non-constructive format of requirements specification into constructive one.

II. REQUIREMENT COVERAGE CHECKING

The procedure of requirement checking is exact sequence of causes and results of some activities (coded with actions,

signals, states), the analysis of which can prove that current requirement is either covered or not. Such checking procedure can be used as a criterion of coverage of specific requirement, i.e. it can become a so-called criteria procedure. In the text below a sequence or "chain" of events will be used for criteria procedure.

Tracking the facts of criteria procedure coverage in system's behavioral scenario (hypothetical, implemented in the model or real system), it can be asserted that the corresponding requirement is satisfied in the system being analyzed.

Procedure of requirement checking (chain) is formulated by providing the following information for all chain elements (events):

- conditions (causes), required for activating of some activity;
- the activity itself, which shall be executed under current conditions;
- consequences – observable (measurable) results of activity execution.

Causes and results are described with signals, messages or transactions, commonly used in reactive system's instances communications [2], as well as with variables states in the form of values or limitations on admissible values. Tracking states' changes, produced by chains activities, lets observe the coverage of corresponding chains.

Problems with unconstructive formulations of requirements are resolved by development of requirement coverage checking procedures on user or intercomponent interfaces.

Thus, chains containing sequences of activities and events can appear as criteria of requirements coverage; in addition, it is possible that criteria of some requirement coverage is specified not with one, but with several chains.

III. INITIAL DOCUMENTS SPECIFYING APPLICATION REQUIREMENTS

In technical documentation each requirement is usually specified in natural language in one of two ways:

- in form of behavioral requirement, when checking scenario (procedure) of requirement coverage checking can be constructed from the text of this requirement in natural language
- in form of non-behavioral requirement, specifying the contents, structure or some desired feature without explanation of how it can be checked.

Formalization of constructively specified requirements is possible together with effective automated analysis of software requirements and is implemented in VRS/TAT technology [3].

In VRS/TAT technology Use Case Maps (UCM) notation [4] (Fig.1) is used for high-level description of the model, while tools for automation of checking and generation work with model in basic protocols language [5].

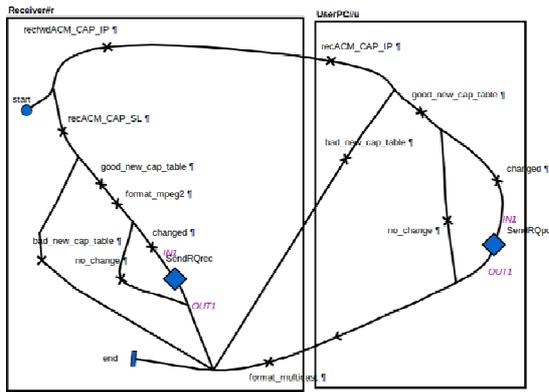


Fig.1 UCM model of two instances: Receiver and UserPC

UCM model (Fig.1) contains two interacting instances model description. Each path on the graph from the event “start” to the event “end” represents one behavioral scenario. Each path contains specified number of events (Responsibilities). Events on the diagram are marked with × symbol, while Stub elements which encode inserted diagram –

Identifier	Requirements	Traceability	Traces
FREQ_GWR.1	Gateway shall transmit ACM_CAP messages repeatedly at an interval of T1 (TBD). This message will carry the ACM Capabilities table.	FREQ_GWR.3	
FREQ_GWR.2	Upon a change in the ACM Capabilities table, the Gateway shall send a new version of ACM_CAP message to the Satellite Terminal.	FREQ_GWR.3	
FREQ_GWR.3	Depending on configuration, ACM_CAP message shall be transmitted either in the MPEG2 private section or in a Multicast message sent across the satellite link.	recACM_CAP_SL recfwdACM_CAP_IP recACM_CAP_IP	FREQ_GWR_3-1 FREQ_GWR_3-2

Fig.2 TRM – Traceability matrix

For example, in the third row of TRM there are 2 chains in “Traceability” column for covering FREQ_GWR.3 requirement. To satisfy the requirement it is enough to trace ACM_CAP signal sending in one of two possible scenarios:

FREQ_GWR.3-1: start, **recACM_CAP_SL**, good_new_cap_table, format_mpeg2, no_chanes, end

FREQ_GWR.3-2: start, **recfwdACM_CAP_IP**, **recACM_CAP_IP**, good_new_cap_table, format_multicast, end.

It should be noted, that during formulating of criteria chains a model of verified functionality is being created which introduces a lot of state variables, types, agents, instances, etc.

V. DEVELOPING INTEGRAL CRITERIA OF REQUIREMENTS COVERAGE

Mentioned above is distinctive feature of VRS/TAT technology – special criteria of each requirement’s coverage checking. Below all criteria related to requirements are listed in ascending order of their strength:

1. Events criterion - coverage level in generated scenarios of subset of events used in criterial chains.
2. Chains criterion - coverage level in generated scenarios of subset of chains (consisting of events and states of variables) with at least one for each requirement.
3. Complex criterion - coverage level in generated scenarios of the whole set of chains specifying

with ♦ symbol. As a result, each scenario contains specified sequence of events. Variety of possible scenarios are specified with variety of such sequences.

In these terms a chain is defined as subsequence of events which are enough to make a conclusion that the requirement is satisfied. A path on the UCM diagram, containing the sequence of events of some chain, is called trace, covering the corresponding requirement. Based on a trace tests can be generated which are needed for experimental evidence of requirement coverage.

IV. TRACEABILITY MATRIX

Verification project requirements formalization starts with Traceability matrix (TRM) [1] creation (TRM for specific project is presented in table format in Fig.2). “Identifier” and “Requirements” columns contain requirement’s identifier, used in the initial document with requirements, and text of the requirement, which shall be formalized. “Traceability” column contains chains of events sufficient for checking of corresponding requirement coverage and “Traces” column – traces or behavioral scenarios used for tests code generation.

integral criteria (combined from criteria 1 and 2) of requirements coverage.

Criteria development shall be adaptive to specific project. Criteria shall be applied flexibly and can be changed according to conditions of scenarios generation.

VI. GENERATING AND SELECTING SCENARIOS WHICH SATISFY TO SPECIFIED INTEGRAL COVERAGE CRITERIA

Trace generation is performed by symbolic and concrete trace generators STG (SymbolicTrace Generator) and CTG (Concrete Trace Generator), which implements effective algorithms of Model Checking. The main problem of trace generation is “explosion” of variants combinations while generating traces from basic protocols, which formalize scenario events, conditions of their implementation and corresponding change of model state after their implementation. Solution here is filtration of generation variants based on numerous limitations specifically defined before trace generation cycle.

There are general and specific limitations. For example, commonly used general limitations are maximum number of basic protocols used in a trace and maximum number of traces generated in a single cycle of generation. Specific limitations are defined by sequences of events in UCM model which guide the process of generation in user-preferable model behavior (so-called Guides). Used are two steps of test scenarios generation by Guides. On the first step guides are created which guarantee specified criteria of system behavior coverage. On the second step guides in UCM notation (Fig.3a) are translated into guides on basic protocols language (Fig.3b) and control trace generation

process. It is important, that only main control points in behavior are specified in guides, while the trace generated from the guide contains detailed sequence of behavioral elements. Such approach to generation significantly reduces the influence of combinatorial explosion on the time of generation during exploring behavioral tree of developed system.

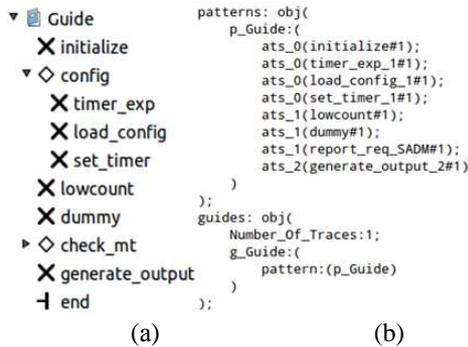


Fig.3. Guides: (a) in UCM notation, (b) in basic protocols language (VRS Guide Language)

VII. AUTOMATIC AND MANUAL PROCESSES OF GUIDES CREATION

There are two possible approaches to guides creation from high-level system description in UCM language: manual and automatic.

Automatic approach allows to generate numerous guides covering system behavior on branches criteria [6]. Each guide contains information about key points of behavioral scenario, starting from initial model state modeled by StartPoint element and ending in final state modeled by EndPoint element. Process of guides generation is performed by UCM to MSC generator [7].

Automatic approach to guides creation can be considered as fast way to obtain test set which satisfies branches criteria, but such approach is not always sufficient. Customer and test engineer may want to check specific scenarios of system behavior for checking some specific requirements. Such scenarios are specified manually by engineer and they are created using UCM Events Analyzer (UCM EVA) tool [8].

In both cases, automatic or manual, problems may occur with guides' coverage by test scenarios.

In automatic mode this is due to the fact that VRS tool not always can successfully generate test scenario form guides because guides are created based on model's control flow and do not consider values of corresponding data flow. At the same time, traces generated based on control flow consider changes in variables values and accordingly model states.

Therefore the actual task is automated analysis of why some guides are not covered by traces and accordingly automation of guides adjusting solved by guides or UCM model modification.

VIII. METHOD OF GUIDES ADJUSTMENT AUTOMATION

Most often reasons of discrepancies are insufficient (not enough detailed) guides specification and mistakes in the sequence of UCM elements in the guide due to incorrect usage of variables, identified as a deadlock on the branch or ramification.

Consider the method of searching of places and reasons of discrepancies between a guide and a trace. Fig.4 shows iterative algorithm of errors searching and fixing automation.

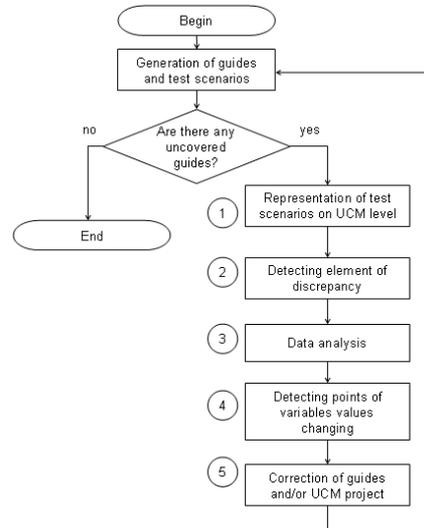


Fig.4. Algorithm of searching and fixing errors in guides (1) Guides and traces generated in VRS are presented as MSC diagrams containing sequences of basic protocols application, thus the first step of the algorithm is mapping basic protocols names on UCM elements.

(2) Comparing the guide and the trace in terms of UCM elements it is possible to define the last trace element which satisfies the sequence of UCM elements specified in the guide. The next uncovered element of the guide will be referred to as the element of discrepancy.

(3) For the element of discrepancy uncovered in the symbolic trace it is possible to explore corresponding data and precondition.

(4) Then variables of precondition shall be singled out into separate list and those places on the UCM diagram where variables of this list are changed shall be analyzed. The analysis shall be performed from the bottom up, starting with the events closest to the element of discrepancy.

(5) After revealing the reason of discrepancy, the guide shall be corrected or the UCM model shall be changed.

The steps above shall be repeated until all guides will be covered by traces.

Consider the process of searching for discrepancy on the example of telecommunication project (Fig. 5).

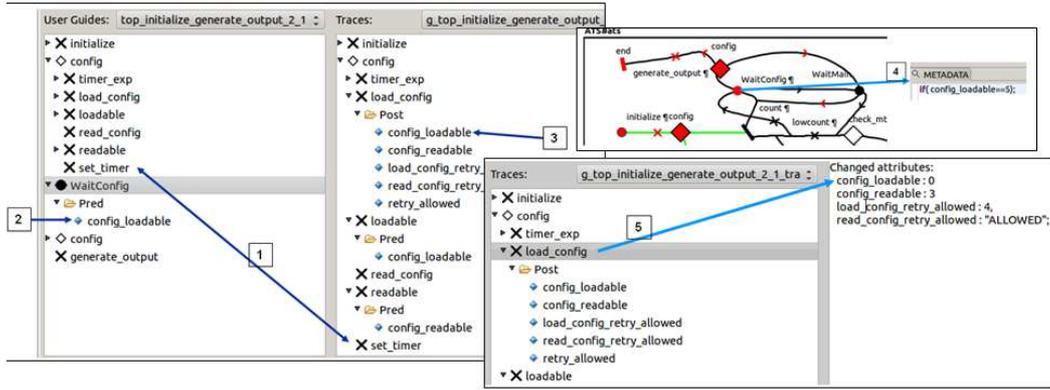


Fig.5. Revealing the reason of discrepancy between the guide and the trace due to variables values

While searching for the reason of guide's non-coverage it is firstly required to find the last element of coincidence between the guide and the trace (1) – **set_timer** in this example. Then the guide element which can not be achieved in the trace (the discrepancy element) shall be found – **WaitConfig** in this example. Analyzing its metadata (2), detect the variable which affects the trace generation (3) and can be the reason of discrepancy - **config_loadable** in this example. After analysis of this variable's values in the UCM model (4) draw a conclusion that in order to apply this element the variable's value shall be 5. The analysis is held only for those points in the trace where **config_loadable** is used. In current case such point is **load_config** element (5), where 0 is assigned to **config_loadable** variable. Thus the conclusion can be made that in order to create a guide which will be successfully covered by a trace it is required to assign value 5 on **load_config** element or change the guide.

Achieved is the reduction of laboriousness in searching for the reasons of errors due to decreasing of the number of points being analyzed which is actual for large industrial projects created in accordance with considered technology.

IX. FORMAL DEFINITIONS OF THE MODEL AND GUIDES LANGUAGE

In [9] the guided search method is described which is used for generation of test scenarios satisfying to specified coverage criteria. Used coverage criteria (Guides) are specified in the form of special regular expressions over the model's transition names alphabet. Guides in abstract way specify the sought-for behaviors in terms of model and simultaneously restrict the number of analyzed states by cutting off the behavioral branches which do not satisfy to specified criteria.

Definition 1. Transitive system M is the tuple $\langle Q, q_0, T, P, f \rangle$, where Q – the set of states, $q_0 \in Q$ – the initial state, T – the set of names of parameterized transitions, P – the set of agents, $f: Q \rightarrow P$ – the map, specifying the actual set of agents in Q state.

To keep connection with initial UCM model of the system being analyzed associate the model's events with the names of its transitions. Parameterization is especially actual for distributed systems composed of parallel asynchronous processes which can be generated and eliminated dynamically. The agent can be presented by one or a set of processes [10].

Definition 2. A path in M from q_i state to q_j state is such sequence of states and transitions

$$q_i \xrightarrow{t_i(a_i)} q_{i+1} \xrightarrow{t_{i+1}(a_{i+1})} q_{i+2} \dots q_j, \text{ that } q_k \in Q \wedge t_k \in T \wedge a_k \in f(q_k) \text{ for each } k \in i..j.$$

Definition 3. A trace in M is the sequence $t_0(a_0), t_1(a_1), \dots, t_n(a_n) \dots$ such that the path $q_0 \xrightarrow{t_0(a_0)} q_1 \xrightarrow{t_1(a_1)} \dots \xrightarrow{t_n(a_n)} q_n \dots$ exists

Definition 4. The language associated with M is denoted as $L(M) \subset L^*$ – this is the set of all traces coming from initial state q_0 .

Definition 5. Guide $a.n$ is the transition a on maximal distance n , which allows the set of traces $\{a, X1a, \dots, X1\dots Xn a\}$, where $X1, \dots, Xn$ – any non-empty symbols from $\{L \setminus a\}$,

$\sim a$ – restriction of a transition, allows any symbol from $\{L \setminus a\}$,

$a; b$ – (where a, b – guides) guides concatenation, allows the set of traces $\{ab\}$,

$a \vee b$ – (where a, b – guides) nondeterministic selection of guides, allows the set of traces $\{a, b\}$,

$a \parallel b$ – parallel composition of the guide a , which represents Za language over the X set alphabet, and the guide b , which represents Zb language over the Y set alphabet, herewith $X \cap Y = \emptyset$, because the sets of agents do not intersect in X and Y . Then the parallel composition of guides is the set represented by $Z^{a \parallel b} = Z_{\uparrow Y}^a \cap Z_{\uparrow X}^b$ language

$\text{join}(a_1, \dots, a_n)$ – the set $\{S_n\}$ of all combinations of guides a_1, \dots, a_n ,

$\text{loop}(a)$ – iteration of the guide a , i.e. $\{aa^*\}$.

The features of the operations listed above are described in details in [11].

Semantically guides specify the “control points” in model's behavior and also specify the criteria (a chain of events in model's behavior) of selection of generated traces for their further usage as test scenarios [12, 13]. Supposed scenarios of modeled system's behavior are checked for admissibility, simultaneously restricting their search.

X. RESULTS OF DEPLOYMENT IN PILOT PROJECTS

Table 1 contains the results of deployment of design and testing integrated technology in the area of wireless telecommunication applications development. As a result, a

significant reduction of laboriousness and increase of software quality were obtained.

Table 1. RESULTS OF TECHNOLOGY DEPLOYMENT IN PILOT PROJECTS.

Project	Number of requirements	Number of basic protocols	Requirements coverage level	Number of found and fixed errors (overall/critical)	Efforts (human-weeks)
Module 1 of wireless network (WN)	400	127	75%	142/11	5.6
Module 2 of WN	730	192	80%	106/18	12
High-level module of WN	148	205	100%	68/23	11
Clients and administrator connection module of WN	106	163	100%	42/8	6
Mobile phone software module	200	170	100%	96/10	7

XI. CONCLUSION

The result of the work is improved integrated technology of verification and testing of software projects which provides:

1. Full automation of industrial software product development process with requirements semantics implementation control.
2. Generation of application's model and symbolic behavioral scenarios, which cover 100% of application's behavioral features.
3. Automated concretization of symbolic traces in accordance with test plan.
4. High level of software development and quality management process automation.

REFERENCES

- [1] S.Baranov, V.Kotlyarov, A.Letichevsky. Industrial technology of mobile devices testing automation based on verified behavioral models of requirements project specifications// «Space, astronomy and programming» – SpbSU, Spb. – 2008. – pp. 134–145. (in Russian)
- [2] Z.Manna, A.Pnueli.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.
- [3] S.Baranov, V.Kotlyarov, A.Letichevsky, P.Drobintsev. The technology of Automation Verification and Testing in Industrial Projects. / Proc. of St.Petersburg IEEE Chapter, International Conference, May 18-21, St.Petersburg, Russia, 2005 – pp. 81-86
- [4] Recommendation ITU-T Z.151. User requirements notation (URN), 11/2008
- [5] A. Letichevsky, J. Kapitonova, A. Letichevsky Jr., V. Volkov, S. Baranov, V. Kotlyarov, T. Weigert. Basic Protocols, Message Sequence Charts, and the Verification of Requirements Specifications. Proc of ISSRE04 Workshop on Integrated-reliability with Telecommunications and UML Languages (ISSRE04:WITUL), 02 Nov 2004: IRISA Rennes France.
- [6] P.Drobintsev, V.Kotlyarov, I.Chernorutsky. Test automation based on user scenarios coverage. “Scientific and technical sheets”, SpbSTU, vol.4(152)-2012, pp.123-126 (in Russian)
- [7] I.Anureev, S.Baranov, D.Beloglazov, E.Bodin, P. Drobintsev, A.Kolchin,, V.Kotlyarov, A. Letichevsky, A. Letichevsky Jr., V.Nepomniashiy, I.Nikiforov, S.Potienko, L.Priyma, B.Tytin. Tools for support of integrated technology for analysis and verification of specifications telecom applications // SPIIRAN proceedings- 2013- №1-28P.
- [8] I.Nikiforov, A.Petrov, V.Kotlyarov. Static method of test scenarios adjustment generated from guides // “Scientific and technical sheets”, SpbSTU, vol.4(152)-2012, pp. 114-119 (in Russian)
- [9] A.Kolchin, V.Kotlyarov, P. Drobintsev. A method of the test scenario generation in the insertion modelling environment // “Control systems and computers”, Kiev: "Akademperiodika", vol.6-2012, pp.43-48 (in Russian)
- [10] A.A. Letichevsky, J.V. Kapitonova , V.P. Kotlyarov, A.A. Letichevsky Jr., N.S.Nikitchenko, V.A. Volkov, and T.Weigert. Insertion modeling in distributed system design // Programming problems. – 2008. – pp. 13–38
- [11] A. Letichevsky Jr., A. Kolchin. Test scenarios generation based on formal model // Programming problems. – 2010. – № 2–3. – pp. 209–215 (in Russian)
- [12] V.P. Kotlyarov. Criteria of requirements coverage in test scenarios, generated from applications behavioral models // “Scientific and technical sheets”, SpbSTU. – 2011. – vol.6.1(138). – pp.202–207. (in Russian)
- [13] Baranov S., Kotlyarov V., Weigert T. Variable Coverage Criteria For Automated Testing. SDL2011: Integrating System and Software Modeling // LNCS. –2012. –Vol.7083. – P.79–89.

MicroTESK: An Extendable Framework for Test Program Generation

Alexander Kamkin*, Tatiana Sergeeva*, Andrei Tatarnikov*[†] and Artemiy Utekhin[‡]

* Institute for System Programming of the Russian Academy of Sciences (ISPRAS)

[†] National Research University Higher School of Economics (NRU HSE)

[‡] Moscow State University (MSU)

Email: {kamkin,leonsia,andrewt,utekhin}@ispras.ru

Abstract—Creation of test programs and analysis of their execution is the main approach to system-level verification of microprocessors. A lot of techniques have been proposed to automate test program generation, ranging from completely random to well directed ones. However, no “silver bullet” has been found. In good industrial practices, various methods are combined complementing each other. Unfortunately, there is no solution that could integrate all (or at least most) of the techniques in a single framework. Engineers are forced to use a number of tools, which leads to the following problems: (1) it is required to maintain duplicating data (each tool uses its own representation of the target design); (2) to be used together, tools need to be integrated (engineers have to deal with different formats and interfaces). This paper proposes a concept of an extendable framework (MicroTESK) that follows a unified methodology for defining test program generation techniques. The framework supports random and combinatorial generation and (what is even more important) can be easily extended with new techniques being implemented as the framework’s plugins.

I. INTRODUCTION

Being extremely complex, modern microprocessors require systematic activities for ensuring their *correctness* and *reliability*. Such activities are usually referred to as *verification* and *testing* [1]. The first of them, verification, is applied in the development stage and focuses on discovering *logical faults* in microprocessor designs (functional faults, interface faults, etc.). The second one, testing, is related to the manufacturing stage and deals with diagnosing *physical faults* in integrated circuits (stuck-at faults, bridging faults, etc.). The general approach to both tasks is based on execution of *verification/test programs*, which are assembly programs causing some *situations* in the design (internal events, component interactions, etc.) [2]. (Throughout the paper we will use the term *testing* to denote both verification and testing.)

By the present time, a great number of techniques for automated test program generation have been proposed. All of them can be subdivided in the following categories: (1) *random generation* [3], (2) *combinatorial generation* [2], (3) *template-based generation* [4] and (3) *model-based generation* [5]. The thing is that there is no a “silver bullet”, which can effectively “fight” against all kinds of testing tasks. In real-life practice, different approaches are used together comple-

menting and strengthening each other. It is a typical solution, for example, when general functionality of a microprocessor is tested by randomly generated programs, while critical logic is verified by advanced model-based techniques.

Unfortunately, there is no framework that could accommodate a variety of test program generation techniques. Engineers have to use a number of tools with different input/output formats, and it is a big problem how to integrate them and keep their configurations in consistent states. It is not difficult to use different tools for solving loosely connected tasks, but settling tightly dependent problems by means of two or more tools might require deep knowledge of their internal interfaces. The root of the problem is that different tools use different representation of the target design, and often one representation is hidden from others. As a result, similar things are specified several times, duplicating data and complicating tests maintenance.

We propose a concept of an *extendable test program generation framework*, named MicroTESK [6]. The idea is to represent knowledge about the microprocessor under test (a *design/coverage model*) in a general way and to provide easy access to that knowledge to a number of test generators built as the *framework’s plugins*. Being shared among various tools, a common model serves as a natural interface for their integration. Moreover, we have unified test generator interfaces, making it possible to use different tools for solving a testing task and even to combine them for doing complex jobs. Interaction between the framework and engineers is done with the help of *test templates* that specify test scenarios in a hierarchical manner (dividing testing tasks into smaller subtasks and linking each of them with an appropriate test generator).

The rest of the paper is organized as follows. Section 2 overviews existing test generation techniques and tools. Section 3 analyses the approaches described in Section 2 and formulates a concept of an extendable test program generation framework. Section 4 outlines the framework architecture and introduces two main components: a *modeling framework* and a *testing framework*. Section 5 considers the modeling framework and its components: a *translator* and a *modeling library*. Section 6 pays attention to the testing framework consisting of a *test template processor*, a *testing library* and a *constraint solver engine*. Section 7 concludes the paper.

This work was supported in part by the Ministry of Education and Science of the Russian Federation under grant #8232 (06/08/2012).

II. TEST PROGRAM GENERATION TECHNIQUES

There is a variety of techniques for test program construction. All of them can be divided into two types: (1) *manual test program development* and (2) *automated test program generation*. Nowadays, manually created tests are rarely used for systematic verification of microprocessors, but the approach is still in use for testing hardly formalizable and highly unlikely “corner cases” in microprocessor behavior. As for automated techniques, they can be subdivided into the following classes: (1) *random generation*, (2) *combinatorial generation*, (3) *template-based generation* and (4) *model-based generation*.

Random generation is the most common technique to produce complex (though unsystematic) test programs for microprocessor verification. Being easy to implement, the method, however, can create a significant workload to the microprocessor and is able to detect some high-quality bugs. One of the most famous generators of that type is RAVEN (Random Architecture Verification Engine) developed by Obidian Software Inc (now acquired by ARM) [3]. To generate test programs, the tool not only applies *randomization*, but takes into account information about common microprocessor faults. RAVEN is built upon pre-developed and custom made modules that can be included into the generator to expand its functionality [3]. Unfortunately, due to the lack of publicly available information, technical details are unclear.

Another approach to test program construction is *combinatorial generation*. A brief analysis of microprocessor errata shows that many bugs can be detected by small test cases (2-5 instructions). Thus, it may be useful to systematically enumerate short sequences of instructions (including *test situations* for individual instructions and *dependencies* between instructions) [2]. The technique has been implemented in the first version of MicroTESK (ISPRAS). The tool supports hierarchical decomposition of a test program generator into *iterators* (each being responsible for iterating its own part of the test) and *combinators* (combining results of the inner iterators into complex test sequences). MicroTESK can also construct test programs with branch instructions (generation is done by enumeration of control flow graphs and bounded depth-first exploration of the execution traces) [7].

The next method is called *template-based generation*. A *test template* is an abstract representation of a test program, where *constraints* are used to specify possible values of instruction operands (instead of concrete values used in usual programs). When constructing a test program, a generator tries to find random solutions to the given constraint systems (such approach is usually referred to as *constraint-based random generation* [8]). Automating routine work, the method considerably increases productivity of engineers. The leading generator of that kind is Genesys-Pro (IBM Research) [4]. This is an architecture-independent tool that uses two types of input data: (1) a *model* (containing architecture-specific information) and (2) *test templates* (describing test scenarios). Genesys-Pro generates test programs in an instruction-wise manner: at each

step, it selects an instruction to be put into a program and, then, formulates and solves a constraint system for the chosen instruction.

As opposed to the previously described approaches, *model-based generation* uses microprocessor models to compose test programs (or test templates). It is worthwhile clarifying the terminology. There are two main types of models used in microprocessor design and test: (1) *instruction-level models* (*behavioral models*) and (2) *microarchitectural models* (*structural models*). Models of the first type specify microprocessors as instruction sets (in other words, they describe a programmer’s view to microprocessors: ‘How to write programs for a microprocessor?’). Models of the second type define the internal structure of microprocessors (this is a computer engineer’s point of view: ‘How a microprocessor is organized inside?’). All test program generation methods and tools apparently use instruction-level models, but only few of them use microarchitectural models. The latter ones are called *model-based*. Let us consider some examples.

In [5], a method for directed test program generation has been proposed. It takes detailed microprocessor specifications written in the EXPRESSION language [9] and translates them into the SMV (Symbolic Model Verifier) description [10]. Specifications define the structure of the design (components and their interconnections), its behavior (semantics of the instructions) and mapping between the structure and the behavior. The key part of the work is a fault model describing typical errors for registers, individual operations, pipeline paths and for interactions between several operations. For each of the fault types, the set of concrete properties is generated, each of which is covered by a test case constructed by SMV (as a counterexample for the negation of the property). The generated test cases are mapped into test programs. As the authors say, the technique does not scale well on complex microprocessor designs. Thus, they suggest using the *template-based* approach as an addition. Test templates are developed by hand and describe sequences of instructions that create certain situations in the design’s behavior (first of all, *pipeline hazards*). Test program generation is performed with the help of the graph model extracted from the specifications. It should be noticed that both approaches are based on rather accurate specifications, and it is better to apply them in the late design stages when the microarchitecture is stable.

In [11], a microprocessor is formally specified as an *operation state machine* (OSM). The OSM is a system of communicating extended finite state machines (EFSMs) modeling the microprocessor at two levels: (1) the *operational level* and (2) the *hardware level*. At the first level, movement of the instructions across the pipeline stages is described (each operation is specified by an EFSM). At the second level, hardware resources are modeled using so-called *token managers*. An operation EFSM changes its state by capturing/releasing tokens of the token managers. The pipeline model is defined as a concurrent composition of the operation EFSMs and resource EFSMs. Test programs are generated on-the-fly by traversing all reachable states and transitions of the joint OSM model.

III. EXTENDABLE FRAMEWORK CONCEPT

Let us analyze the test generation techniques and tools having been surveyed previously and formulate a concept of an extendable test program generation framework. It is worthwhile answering the questions: ‘What is *extendability*?’ and ‘What is an *extendable* framework?’ In general terms, *extendability* is a framework characteristic that shows how much effort it takes to integrate a new or existing component (in our case, a microprocessor model or a test generation engine) into the framework. Indeed, the less effort it is required, the more extendable the framework is. The object of this study is to suggest a framework architecture that would minimize the effort for creating new models/engines and plugging them into the framework.

There is a number of basic requirements that all kinds of extendable frameworks are expected to comply with. A system should be architected in such a way that there is a *core (platform)* and there are *extensions (plugins)* connected to the core via the *extension points*. Obviously, there should be well-defined interfaces between the core and its extensions as well as clear mechanisms for installing extensions into the framework and calling them for solving particular tasks. An optional requirement, which, we think, is essential for true extendable frameworks, is *open source*. The open source paradigm significantly simplifies creation and distribution of framework extensions.

There are also specific requirements to test program generation frameworks. Analyzing the approaches presented in the previous section, we can see that all of them use *instruction-level models* (either explicitly or implicitly). Evidently, to create a valid test program, one should know *instruction formats* and *instruction preconditions*. However, if more sophisticated programs need to be generated, more complicated models should be utilized (finite state machines, nets, etc.). In our opinion, the core should be formed around instruction-level models, while more specialized models/engines should be organized as framework extensions. Another suggestion is to divide the test program generation framework into two parts: (1) the *modeling framework* and (2) the *testing framework*, each having its own core and being extendable.

The core of the modeling framework allows describing *registers* (as variables storing fixed-size bit vectors), *memory* (as an array of machine words) and *instructions* (as atomic operations over registers and memory). More detailed specification is provided by so-called *model extensions*. There is a number of points that such extensions can be connected to (e.g., the memory access handler and the instruction execution handler). The framework supports standard extensions for specifying *memory management* (cache hierarchy, address translation mechanisms, etc.) and *pipelining* (interconnection between pipeline stages, control flow transfers, etc.). Note that the standard extensions, in turn, have extension points and can be easily customized by engineers (e.g., it is possible to define a cache replacement strategy or describe behavior of a pipeline stage).

The testing framework is comprised of engines of two types: (1) *test sequence generators* and (2) *test data generators*.

The main test sequence generators are *random* and *combinatorial generators* [2], [3]. It is explained by the fact that the modeling framework is organized around instruction-level models, which give no information on how to compose instructions into sequences for achieving particular testing goals. A useful feature is support for *test program composition* [12]. Given two test programs (or test templates) focusing on different (and more or less independent) situations, it may be interesting to shuffle them with the intention to cause situations to occur simultaneously or close to each other. More advanced *model-based generators* can be installed into the framework together with the corresponding models. Moreover, extension of the modeling framework should be always accompanied by the extension of the testing framework (if a new type of models is added into the framework, one should describe how to generate tests on the base of such models).

We think, it is a promising practice to use *constraint solvers* for test data generation (as it is done in Genesys-Pro [4]). It implies that test situations are expressed as *constraints* on the instruction operands and the microprocessor state. An important property of the approach is that situations can be easily combined by conjuncting the constraints. In contrast to Genesys-Pro, we suggest using general-purpose SMT solvers (like Yices [13] and Z3 [14]) supporting the unified SMT-LIB notation [15]. In addition, the generation core can be extended with *custom generators* (which are useful when situations are hardly expressible in terms of constraints). There is also a library of predefined generators including *random generators* and *directed generators* (e.g., for the floating-point arithmetic [16]).

IV. MICROTESK FRAMEWORK ARCHITECTURE

The MicroTESK framework is divided into two main parts: (1) the *modeling framework* and (2) the *testing framework*. The purpose of the modeling framework is to represent a model of the microprocessor under test (a *design model*) as well as model-based testing knowledge (a *coverage model*). The design/coverage model is extracted from *formal specifications* written in an *architecture description language (ADL)*. The testing framework, for its turn, is responsible for generating test programs for the target microprocessor on the base of information provided by the model. Testing goals are defined in *test templates* written in a *template description language (TDL)*.

The MicroTESK modeling framework consists of (1) a *translator* (analyzing formal specifications in an ADL and producing the microprocessor model) and (2) a *modeling library* (containing interfaces to be implemented by a model and classes to be used as building blocks) (see Figure 1). The translator includes two back-ends: (1) a *model generator* (constructing an executable design model) and (2) a *coverage extractor* (building a coverage model for the microprocessor instructions). In a similar fashion, the modeling library is split into *design* and *coverage libraries*.

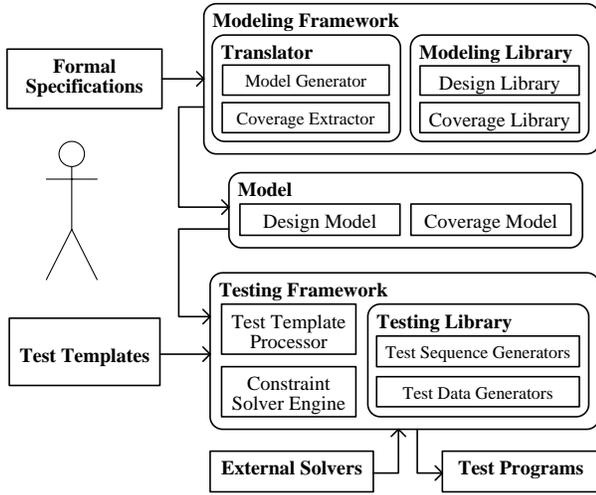


Figure 1. General structure of the MicroTESK framework

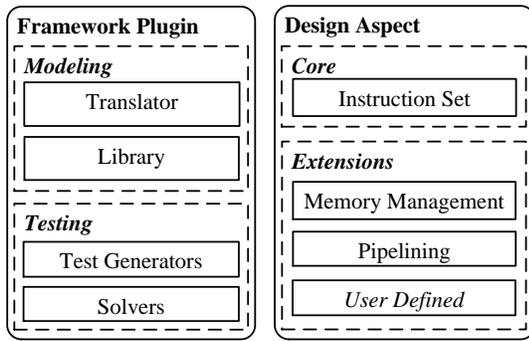


Figure 2. Design aspects and organization of a MicroTESK plugin

Components of the MicroTESK testing framework are as follows: (1) a *test template processor* (handling test templates written in a TDL and generating test programs), (2) a *testing library* (containing a wide range of *test sequence generators* and *test data generators* used by the test template processor) and (3) a *constraint solver engine* (providing the test generators of the testing library with a Java interface to external *SMT solvers*).

The framework components are not monolithic – they include some *core* functionality as well as *extensions* oriented to specific tasks. Such tasks are usually grouped according to the *design aspects* they deal with. All extensions related to the same aspect are united into a *framework plugin* (see Figure 2). To extend the framework with features aimed at modeling/testing a new design aspect, one should develop extensions for all of the framework components, including a *modeling library* (to provide building blocks for modeling the design aspect), a *testing library* (to let the framework know of how to test the design aspect) and a *specification language* coupled with a *translator* (to make it possible to express properties on the design aspect in a human-readable form).

The framework functionality is divided into the following aspects: (1) *instruction set*, (2) *memory management* and (3) *pipelining*. Forming the framework core, the first aspect is responsible for modeling microprocessor instructions and generating test programs on the base of the instruction-level models (random, combinatorial and template-based generators are in active use). Support for the next ones, memory management and pipelining, is implemented in the standard plugins. For other design aspects, custom plugins can be created and installed into the framework.

V. MICROTesk MODELING FRAMEWORK

The purpose of the MicroTESK modeling framework is to represent knowledge about a microprocessor and share that knowledge with the testing framework. An engineer provides *formal specifications* of the microprocessor under test. The specifications are processed by the *translator* (including the front-end and two back-ends, the *model generator* and the *coverage extractor*). The translator produces the *model* by using the *modeling library* (comprising the *design* and *coverage libraries*). Let us consider the modeling framework components in more detail.

A. Translator

The *translator* processes *formal specifications* of the microprocessor and builds the *design/coverage model* (applying the model generator/coverage extractor and using the building blocks defined in the design/coverage library). Note that microprocessor specifications are written in a mixture of languages, each being responsible its own design aspect. The central part of specifications is related to the instruction set architecture; other parts describe memory management, pipelining, etc. As specifications are heterogeneous, the translator is actually represented as a set of tools processing their parts of specifications.

At the moment, Sim-nML [17], [18] is the only ADL supported by MicroTESK for specifying microprocessors at the instruction level. Sim-nML code specifying the integer addition instruction (ADD) from the MIPS instruction set [19] is shown below. Several things need to be emphasized: (1) specifications can use the predefined function UNPREDICTABLE to indicate the situations, where the design's behavior is undefined; (2) by analyzing control and data flows in instruction specifications one can automatically extract the coverage model; (3) instructions can be grouped together providing the framework with useful information to be used within test templates; (4) basing on such specifications, the framework is able to predict the result of the test program execution [12].

```

op ADD(rd: GPR, rs: GPR, rt: GPR)
  action = {
    if (NotWordValue(rs) || NotWordValue(rt))
      then
        UNPREDICTABLE();
    endif;
    tmp = rs <31..31>::rs <31..0> +
          rt <31..31>::rt <31..0>;
  }

```

```

if (tmp<32..32> != tmp<31..31>)
then
  SignalException("IntegerOverflow");
else
  rd = sign_extend(tmp_word<31..0>);
endif;
}

```

```

syntax = format("add %s, %s, %s",
  rd.syntax, rs.syntax, rt.syntax)

```

op ALU = ADD | SUB | ...

B. Design Library

The *design library* is intended to represent a microprocessor model, which is used to simulate instruction execution and to keep track of the design state during test program generation. State tracking is essential for generating *self-checking tests* (i.e., programs with built-in checks of the microprocessor state). In addition to the *instruction simulator* (which simulates instructions and updates the model state) and the *state observer* (which provides access to the model state), the design model provides *meta-information* describing the design elements (registers, memory, instructions, etc.). The meta-information is the main interface between the modeling framework and the test template processor.

The design library has several *extension points* that allow engineers to connect their components. The set of extension points includes (1) the *memory access handler* and (2) the *instruction execution handler*. The handler of the first type is invoked every time a memory location is accessed for reading or writing. It may encapsulate memory management logic such as address translation and caching. The handler of the second type is launched when an instruction is executed. It is usually used to model the microprocessor pipeline – decomposition of instructions into microoperations and their scheduling.

C. Coverage Library

The *coverage library* is used to describe situations that can occur in a microprocessor (an overflow, a cache miss/hit, a pipeline bypass, etc.). Such a description (referred to as a *coverage model*) serves as a basis for generating test programs (especially, for creating test data for individual instructions of a program). Besides the *test situations*, the coverage model contains *grouping rules*, classifying microprocessor instructions according to some criteria (number of operands, resources being accessed, control flow structure, etc.). Similar to the design model, the coverage model provides meta-information on its elements, which is used by the test template processor.

Each test situation has a unique name that can be used in a test template to refer to the situation. There is a mapping of situation names onto test generators. Thus, the test template processor knows which engine to use to create a particular test case. To make the engine comprehend how it can be done, the situation include an engine-specific description of the condition/action causing the situation to occur. The most

usable engine built-in into the framework uses the constraint-based description of situations and constraint solving [20].

VI. MICROTESK TESTING FRAMEWORK

The MicroTESK testing framework is responsible for generating test programs. An engineer provides a *test template* describing a test scenario for the microprocessor under test. The test template is handled by the *test template processor* by using the *engines* of the *testing library*: (1) it applies the *test sequence generators* to construct a *symbolic test program* (i.e., sequence of instructions annotated with test situations); (2) it requests the *test data generators* to generate concrete values of the instruction operands; (3) it instantiates the test program by inserting *control code* initializing the registers and the memory with the generated test data. Let us consider the testing framework components in more detail.

A. Test Template Processor

The *test template processor* is a runtime environment that handles a *test template*, chooses appropriate engines of the *testing library* and produces a *test program*. The supported TDL is organized as a Ruby [21] library. It allows describing instruction sequences in a way it is done in the assembly language (by using the *meta-information* provided by the *design model*) though supporting high-level scenario description constructs. The latter ones can be subdivided into two types: (1) *native Ruby constructs* (conditional statements, loops, etc.) and (2) *special MicroTESK constructs* (test sequence blocks, test situations, etc.). A simple test template example is given below.

```

# Assembly-Style Code
add r[1], r[2], r[3]
sub r[1], r[1], r[4]

# Ruby Control Statements
(1..3).each do |i|
  add r[i], r[i+1], r[i+2]
  sub r[i], r[i], r[i+3]
end

# Test Sequence Block
block (:engine => "random",
  :count => 2013)
{
  add r[1], r[2], r[3]
  sub r[1], r[2], r[3]
  # Test Situation Reference
  do overflow end
}

```

An important notion used in test templates is a *test sequence block*. In fact, a test template is a hierarchical structure of test sequence blocks, each holding a set of instructions (or nested blocks) and specifying a *test sequence generator* (and its parameters) to be used to produce a test sequence. The test template processor constructs test sequences for the

nested blocks by applying the corresponding engines and then *combines/composes* the built sequences with the root engine (an example is given the section “*Test Sequence Generators*”).

Another important feature of the test template processor is support for generation of *self-checking tests*. When constructing a test program, the test template processor can inject special pieces of code that check whether the microprocessor state is valid in the corresponding execution point. Such code (called a *test oracle*) compares data stored in the previously accessed registers and memory blocks with the reference data (calculated by the *instruction simulator*) and terminates the program if they do not match.

B. Test Sequence Generators

A *test sequence generator* is organized as an *iterator* of test sequences. In the simplest case, a test sequence generator returns a single test sequence for a single test sequence block. As blocks can be nested, generators can be *combined/composed* in a recursive manner. To do it, two strategies should be defined for each non-terminal block: (1) a *combinator* (describing how to combine the results of the inner iterators) and (2) a *compositor* (defining the method for merging several pieces of code together). Thus, a combinator produces the combinations of the inner test sequences, while a compositor merges those sequences into the one.

The *testing library* contains a variety of combinators and compositors. The most usable combinators are: (1) a *random combinator* (produces a number of random combinations of the inner iterators’s results), (2) a *product combinator* (creates all possible combinations of the inner blocks’ test sequences) and (3) a *diagonal combinator* (synchronously requests the inner iterators and joins their results). The set of implemented compositors include: (1) a *random compositor* (randomly mixes the inner test sequences), (2) a *catenation compositor* (catenates the inner test sequences) and (2) a *nesting compositor* (embeds the inner test sequences one into another). Note that engineers are allowed to add their own test sequence generators, combinators and compositors into the testing library and invoke them from test templates. Let us consider a simple example.

```
# Test Sequence Block
block (:combine => "product",
       :compose => "random") {

# Nested Block A
block (:engine => "random",
       :length => 3,
       :count => 2) {
  add r[a], r[b], r[c]
  sub r[d], r[e], r[f]
  mult r[g], r[h]
  div r[i], r[j]
}
}
```

```
# Nested Block B
block (:engine => "permutate") {
  ld r[k], r[l]
  st r[m], r[n]
}
}
```

In the example above, there is one top-level block containing two nested blocks, A and B. Block A consists of four instructions, ADD, SUB, MULT and DIV. Block B consists of LD and ST. The engine associated with A generates two sequences (:count => 2) of the length three (:length => 3) composed of the instructions listed in the block. The engine associated with B generates all permutations of the inner instructions (there are two permutations of two elements). The top-level engine produces all possible combinations of the nested blocks’ sequences (:combine => "product") and randomly mixes them (:compose => "random"). The result may look as follows.

```
# Combination (1,1)
sub r[d], r[e], r[f] # Block A
ld r[k], r[l] # Block B
div r[i], r[j] # Block A
st r[m], r[n] # Block B
add r[a], r[b], r[c] # Block A

# Combination (1,2)
st r[m], r[n] # Block B
sub r[d], r[e], r[f] # Block A
ld r[k], r[l] # Block B
div r[i], r[j] # Block A
add r[a], r[b], r[c] # Block A

# Combination (2,1)
mult r[g], r[h] # Block A
mult r[g], r[h] # Block A
ld r[k], r[l] # Block B
add r[a], r[b], r[c] # Block A
st r[m], r[n] # Block B

# Combination (2,2)
mult r[g], r[h] # Block A
st r[m], r[n] # Block B
mult r[g], r[h] # Block A
ld r[k], r[l] # Block B
add r[a], r[b], r[c] # Block A
```

C. Test Data Generators

A symbolic test program produced by *test sequence generators* does not necessarily define values of all of the instruction operands (leaving some of them either undefined or deliberately ambiguous). The job of *test data generators* is to construct operand values on the base of the provided test situations. Test data generation relies on the *constraint solver engine* that constructs operand values by solving the corresponding constraints. To achieve a given test situation,

the *test template processor* selects an appropriate test data generator and requests the *design model* for the state of the involved design elements. After that, it initializes the closed variables of the constraint (variables whose values are defined by the previously executed instructions) and calls the constraint solver engine to construct the free variables' values.

As soon as the operand values are constructed, the test data generator returns *control code*, which is a sequence of instructions that accesses the microprocessor resources associated with the instruction operands and brings them into the required states. For example, if an instruction operand is a register, control code writes the constructed value into that register. Following the concept of the *constraint-based random generation*, different calls of a test data generator may lead to different values of free variables. However, each generated set of values should cause the specified test situation.

D. Constraint Solver Engine

The *constraint solver engine* is a framework component that helps *test data generators* to construct *test data* by solving *constraints* specified in *test situations*. The engine is implemented as a collection of *solvers* encapsulated behind a generic interface. Solvers are divided into two major families: (1) *universal solvers* (handling a wide range of constraint types) and (2) *custom solvers* (aimed at specific test data generation tasks).

Universal solvers are built around external *SMT solvers* (like Yices [13] and Z3 [14]), which provide a rich constraint description language (supporting Boolean algebra, arithmetic, logic over fixed-size bit vectors and other theories) as well as effective decision procedures for solving such constraints. The MicroTESK framework uses Java Constraint Solver API [20] providing a generic interface to SMT-LIB-based constraint solvers [15]. The library allows dynamically creating constraints in Java, mapping them to the SMT-LIB descriptions, launching a solver and transferring results back to Java.

Some test situations are hardly expressible in terms of SMT constraints (e.g., situations in floating-point arithmetic, memory management, etc.). For such situations engineers are able to provide special custom solvers/generators. Note that custom solvers can also use SMT solvers to construct test data; though they usually implement non-trivial logic on forming a constraint system and interpreting its solution. When the design/coverage model is extended with a new type of knowledge, it often means a need to provide a corresponding custom solver. To facilitate extension of the constraint solver engine with new solvers, both universal and custom solvers implement uniform interfaces.

VII. CONCLUSION

We have suggested the extendable architecture for test program generation framework. The proposed solution, named MicroTESK, can combine a wide range of microprocessor modeling and testing techniques. The central part of the framework is built around instruction-level models and random/combinatorial test program generators. More complicated

types of models and test generation engines are supposed to be added as the framework's extensions. The goal of our work is not to create a "silver bullet" for microprocessor verification and testing (which, we believe, does not exist), but to organize a flexible, open-source environment being able to absorb a variety of useful approaches. Let us emphasize that the development having been launched at ISPRAS is based on the many-years experience of verifying industrial microprocessors. The work has not been finished, and there are a lot of things need to be done. In the nearest future, we are planning to implement the framework core and customize the generator for widely-spread microprocessor architectures, including ARM and MIPS. We are also working on MicroTESK's extensions for specifying/testing memory management mechanisms and pipeline control logic.

REFERENCES

- [1] M.S. Abadir, S. Dasgupta, *Guest Editors' Introduction: Microprocessor Test and Verification*. IEEE Design & Test of Computers, Volume 17, Issue 4, 2000, pp. 4–5.
- [2] A. Kamkin. *Test Program Generation for Microprocessors*. Institute for System Programming of RAS, Volume 14, Part 2, 2008, pp. 23–63 (*in Russian*).
- [3] http://www.arm.com/community/partners/display_product/rw/ProductId/5171/.
- [4] 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, Volume 21, Issue 2, 2004, pp. 84–93.
- [5] P. Mishra and N. Dutt. *Specification-Driven Directed Test Generation for Validation of Pipelined Processors*. ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 13, Issue 3, 2008, pp. 1–36.
- [6] <http://forge.ispras.ru/projects/microtesk>.
- [7] A. Kamkin. *Some Issues of Automation of Test Program Generation for Branch Units of Microprocessors*. Institute for System Programming of RAS, Volume 18, 2010, pp. 129–150 (*in Russian*).
- [8] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus and G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification*. AI Magazine, Volume 28, Number 3, 2007, pp. 13–30.
- [9] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt and A. Nicolau. *EXPRESSION: An ADL for System Level Design Exploration*. Technical Report 1998-29, University of California, Irvine, 1998.
- [10] <http://www.cs.cmu.edu/~modelcheck/smv.html>.
- [11] T.N. Dang, A. Roychoudhury, T. Mitra and P. Mishra. *Generating Test Programs to Cover Pipeline Interactions*. Design Automation Conference (DAC), 2009, pp. 142–147.
- [12] A. Kamkin, E. Kornychin and D. Vorobyev. *s Reconfigurable Model-Based Test Program Generator for Microprocessors*. Software Testing, Verification and Validation Workshops (ICSTW), 2011, pp. 47–54.
- [13] B. Dutertre and L. Moura. *The YICES SMT Solver*. 2006 (<http://yices.csl.sri.com/tool-paper.pdf>).
- [14] L. Moura and N. Bjørner. *Z3: An Efficient SMT Solver*. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.
- [15] D.R. Cok. *The SMT-LIBv2 Language and Tools: A Tutorial*. GrammaTech, Inc., Version 1.1, 2011.
- [16] M. Aharoni, S. Asaf, L. Fournier, A. Koifman and R. Nagel. *FPgen – A Test Generation Framework for Datapath Floating-Point Verification*. High Level Design Validation and Test Workshop (HLDVT), 2003. pp. 17–22.
- [17] M. Freericks. *The nML Machine Description Formalism*. Technical Report, TU Berlin, FB20, Bericht 1991/15.
- [18] R. Moona. *Processor Models For Retargetable Tools*. International Workshop on Rapid Systems Prototyping (RSP), 2000, pp. 34–39.
- [19] *MIPS64TM Architecture For Programmers*. Volume II: The MIPS64TM Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.
- [20] <http://forge.ispras.ru/projects/solver-api>.
- [21] <http://www.ruby-lang.org>.

Probabilistic Networks as a Means of Testing Web-Based Applications

Anton Bykau

Department of Informatics
Belarusian State University of Informatics and Radiotechnics
Minsk, Republic of Belarus
anton.bukov@gmail.com

Abstract— The article describes the mechanism used to control GUI tests coverage and the technique of GUI application under test model building using probabilistic networks. The technology of combining GUI tests into the common network has been developed. The mechanism to report defects is proposed.

Keywords— *probabilistic network testing; web interfaces; automation*

I. INTRODUCTION

Testing is a process of execution of the program to detect defects [1]. The generally accepted methodology for the iterative software development Rational Unified Process presupposes the performance of a complete test on each iteration of development. The testing process of not only new but also earlier code written during the previous iterations of development, is called regression testing. It's advisable to use the automated tools when performing this type of testing to simplify the tester work. "Automation is a set of measures aimed at increasing the productivity of human labor by replacing part of this work, the work of machines". [2] The process of automation of software testing becomes part of the testing process.

The requirements formulation process is the most important process for software developed. The V-Model is a convenient model for information systems developing. It's become government and defense projects standard in Germany. [3] The basic principle of V-model is that the task of testing the application that is being developed should be in correspondence with each stage of application development and refinement of the requirements. One of the development model challenges is the system and acceptance testing. Typically, this type of testing is performed according to the black box strategy and is difficult for automation because automated tests have to use the application interface rather than API.

"Capture and replay" is the one of the most widely used technologies for web application test automation according to the black box strategies today [4]. In accordance with this technology the testing tool records the user's actions in the internal language and generates automated tests.

Practice shows that the development of automated tests is most effective if it is carried out using modern methods of

software development: it is necessary to analyze the quality of the code, merge into the library the duplicate code of tests, which must be documented and tested. All this requires a significant investment of time and the tester should have the skills of the developer.

Thus, the question arises of how to combine the user actions recording technology and the manually automated tests development, how to organize the automated tests verification, and whether it is possible to develop an application and automated tests in parallel according to the methodology of the test-driven development (TDD).

There are systems capable of determining the set of tests that must be performed first. Such systems offer manually associate automated tests with the changes in the source files of application under test. However, the connection between the source and the tests can be expressed in terms of conditional probabilities. The probabilistic networks used in the artificial intelligence, could also be useful when defining the relations automatically based on the statistics of tests results. By using probabilistic networks we can link interface operations and test data and this will allow reducing the complexity of automation.

II. KEY ELEMENTS OF PROPOSED TESTING TECHNOLOGY

For tests automation we could use a probabilistic network that has the following structure:

The first level network shown in Fig. 1, consists of two layers, which determine the location of graphical controls on the web page. Top-level nodes Fig. 1.1 are either pages or the condition of the tested application page such as a page of the user authentication. Lower-level units are templates used to identify GUI elements Fig. 1.3. Some nodes are GUI container templates Fig. 1.3. Fig. 1.4 shows the properties of the selected node, like the template for the password field. Graphic elements that occur more than on one page can be transferred to a general unit for multiple pages, such as Fig. 1.5 that shows the menu items. Fig. 1 shows only the network connection between the unit and the common elements of the page to simplify the visualization of the network for testers.

The availability of GUI templates and states of the web interface allows monitoring the test coverage for interface of

application with tests; it also allows to effectively adapt automated tests to new versions of the tested application.

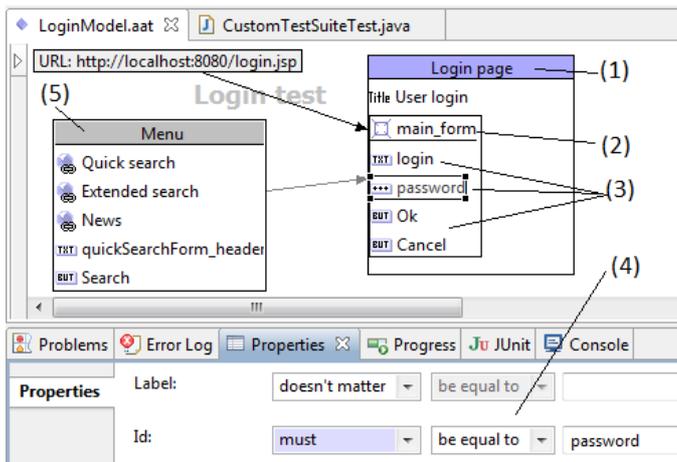


Fig. 1 GUI elements composition

The main goal of the second level network is to describe the workflow of the program in the form of interconnected rules, describing the program states and GUI interface actions (see Fig. 2). The network consists of two layers and two types of nodes that include the nodes of all possible states of the program (see Fig. 2.1) and the nodes of all possible program actions (see Fig. 2.2). The communication network describes the state transitions as a result of GUI activities. The page can be linked to the data (see Fig. 2.3) to describe the state of the page containing dynamic elements, for example, a table with a date. The data layer consists of nodes storing the state of the tested application and the operations that modify the data. Fig. 2.3 describes the results table which is used in Fig. 2.4. Each table row should include a reference to additional information; the lower part of the table should contain additional 3 references (see Fig. 2.4) while the search box should include the search phrase (see Fig. 2.5). The state of some graphical elements is not preserved in the data layer (Fig. 2.6) to simplify the automation process.

The system of tests automation constantly analyzes the state of the application interface during the tests recording time. If the same sequence of actions is repeated many times, the system offers to merge this sequence for multiple pages into a common block (see Fig. 1.5). The recorded actions and states will not be duplicated. When writing the second and subsequent tests, the system adds only unknown conditions and operations. Although the model interface can be split into separate files, it will not prevent the system from linking blocks common for several pages. Often, automated tests complicate the process of automation as a result of an unsuccessful candidate decomposition code. A single model of the whole test interface can help to avoid duplication and to refactor the source of recorded tests.

The system determines an appropriate relationship between the states if a previously unknown combination of actions was done between the known conditions in the process of test recording.

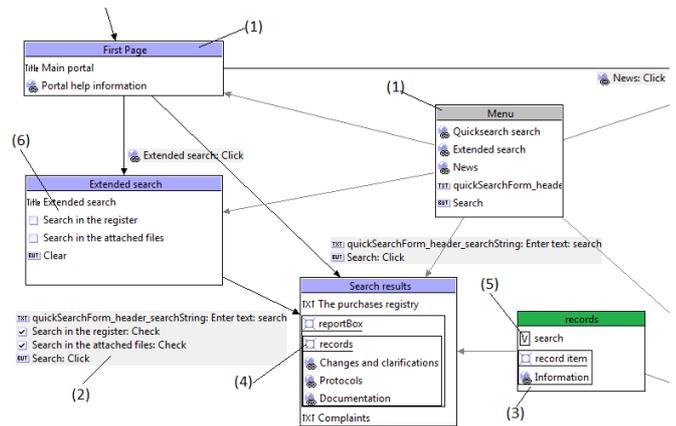


Fig. 2 Program Algorithm

The third level network describes the tests and defects of the tested program. The top layer describes a set of written tests (see Fig. 3.1) and is connected to the nodes pages (see Fig. 3.2). Each test case describes what action and what graphics should be checked (Fig. 3.3). Subsequently, the system will find preliminary steps for testing, using an algorithm to find a way to graph states proposed by S. Russell [5] to perform one or more tests.

The relationship between the test and page nodes can be divided by a bug note to describe the defect (see Fig. 3.4). The defect can be in one of the following states turning a positive test into a negative one (see Fig. 3.5):

- presence of an undocumented and uncorrected defect (the node is absent)
- expectance of an uncorrected and described defect (the defect node created and verify defect reproduce)
- absence of the expected defect (the defect node can't reproduce the defect)
- confirmed lack of the described defect (the defect note verifies the defect absence)

The test system displays test results in a different way for developers and testers. This allows evaluating the correctness of the automated tests and independently assessing the quality of the tested application. The presence of the life cycle of a defect integrates accounting system defects and automated testing.

The priority value is associated with each test node. This characteristic is actually the probability that the test result will be incorrect, for example, the bug will not be reproduced or the expected page will not load properly. The higher the probability of the failure, the more important it is to run the test to fix the problem and increase the stability of testing.

The priority of the test run can be set manually by the tester, or can be statistically calculated on the basis of the associated defect status changes, or the associated source code changes, or on the basis of the results of the same test for the same controls of other pages. Typically, these tests are associated with blocks of common elements (see Fig. 3.6).

The most important testing task is to measure the relatedness of the test results from the internal state to the

application, or previous operation. The main problem of such measurements is an extremely large number of conditions with should be measured by the test system. The whole history of the automated testing system is preserved, and each performed activity is associated with a corresponding network node .

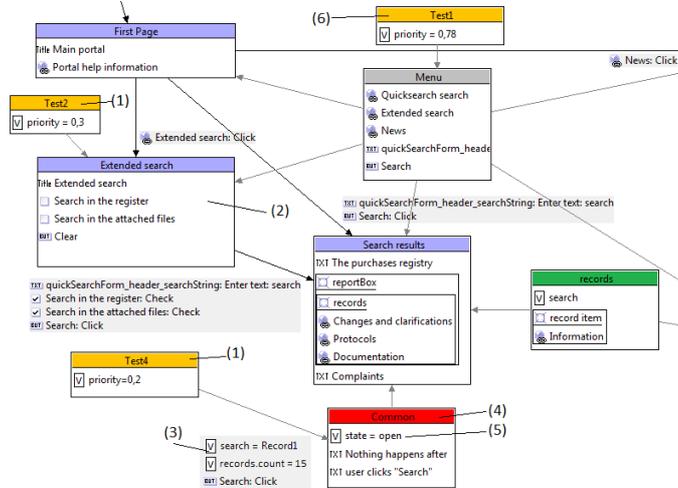


Fig. 3 Description of Tests and Defects

The fourth level network describes the knowledge about of testing purposes (see Fig. 4). The network consists of the nodes which represent the testing goal (see Fig. 4.1) and is associated with one or more tests (see Fig. 4.2). The example of the target can either be one or a group of pages and of the tested interface program (see Fig. 4.3).

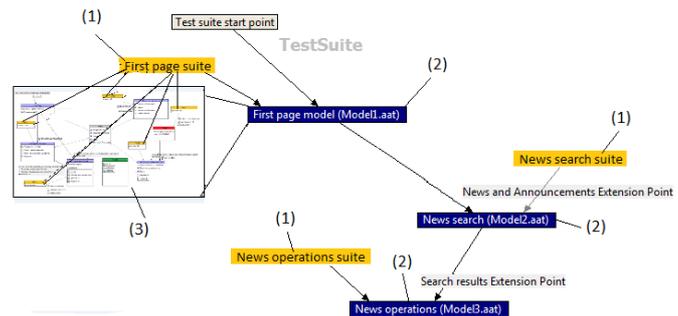


Fig. 4 Description of Test Purposes

Two algorithms are used for the network work; they are the calculation network algorithm and the path finding algorithm. The calculation algorithm determines the status of the tested application using patterns of GUI elements, and calculates the priority of tests running, analyzing what associated source files have been changed and what defects have been fixed. The path finding algorithm finds the sequence of preparatory steps to perform the test in order to select a sequence of tests that will allow to reduce the total test time.

III. NETWORKS CALCULATION ALGORITHM

The test system uses a modification of the Bayesian networks calculation algorithm proposed by R. Schechter [6]. The modified algorithm can calculate the network even in the presence of the following features:

- Probabilistic network links can be directed or undirected.
- Probabilistic network links can contradict each other.

The first level network must be recalculated, despite the controversy because the program interface can be wrong: the graphic elements may not work properly, requirements may be outdated or the tester can make mistakes. The goal of the test system is to detect these mistakes.

Probabilistic networks nodes can take multiple values which are characterized by probabilities. The probability evaluate whether the node actually takes this particular probability value. The condition corresponding to the node, its condition is called a characteristic. The sum of all characteristics of the multivalued node equals 1.

$$P(A1)+P(A2)+...+P(An)=1 \tag{1}$$

The network connection may be contradictory. Contradictions arise when there is a problem in the test program. The algorithm has to consider the mutual influence of links and to make approximation solutions. On the other hand, the system can independently adjust its work in case of the loss of control of the tested application.

To describe the algorithm we shall present an example of calculating the characteristics of the two states of simple networks. For simplicity, we use only the connections between two nodes while the binary characteristics and the conditional probabilities equal 1 or 0. We shall use Bayes' formula to calculate the characteristic of the required node:

$$P(A)=P(A/B)*P(B) \tag{2}$$

Let's consider an example where the communication is in conflict. Let's suppose that we know that:

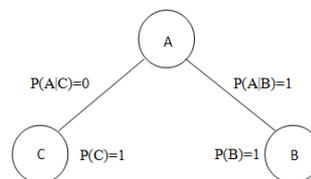


Figure 5. Contradictory Conditions

When looking at Figure 5 we can consider connections C-A and B-A independent, and the probability node A is calculated as the probability of two independent events:

$$P(A)=(P(A/B)*P(B)+P(A/C)*P(C))/2 \tag{3}$$

Another difficulty is the presence of cycles in the network. Let's add to the previously described structure of the network Figure 5 connection C-B, and calculate the values of the characteristics B and C on the basis of the given vertex A

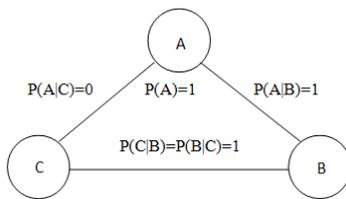


Figure 6. Contradictory Dependencies

When looking at the network (Figure 6) we can see an apparent contradiction: the links from node A assign different states to nodes B and C but the link C - B requires the identity of node values.

We could solve the contradiction by reducing the trust in relations of the network but we can't do that until we know the correct values. The temporary solution should be the construction of the set of the skeletons of trees of a network for any given performance with equal confidence in relations and the known value of the node A. There are three skeletons for the network (see Figure 6). It's easy to calculate the probability value of the nodes for each such skeleton. Finally we find the average value for each characteristic for each skeleton tree. The solution can be presented in the following way:

$$P(C=1)=P(B=0)=0,333, P(C=0)=P(B=1)=0,667 \quad (4)$$

The advantage of the algorithm is that the connection can combine more than two characteristics and the logic of the relationships conversions can be defined by the programmer manually. The link may be represented as a function of several variables that return the value to the node to which it is directed and that can be defined in any programming language. The presence of a double direction link between the two characteristics can be described by two oppositely oriented links.

IV. AUTOMATION PROCESS

The probabilistic network for the application testing can be created on the basis of the "record and play" tool. This method is useful when the testing system has a poor knowledge of the tested application. When recording the test system stores the sequence of the application states and interface actions. After the recording of the test the test automation system invites the tester to answer some questions. The recorded net diagram of transitions between the states should become the result of the recording.

The tester creates a test node and describes the data need for the test to define the test case. He can create a set of tolerance values for each GUI element of the page (see Fig. 2.3). In this case, it will reach the coverage criterion according of the black box strategy "covering the tolerance range", based on the testing criteria of the class of input and output data.

The network for the application testing can be created using the answers to the questions about the interface. This interface is effective when the model contains enough knowledge about the tested program. The system will be testing the application

in the background, and if there is a problem, it will ask the tester without stopping the execution of other tests.

The system operation and the work of the tester start with some initial page and state of the tested application. This condition is evaluated and if the condition does not correspond to GUI templates, the system will suggest that we add a new state to the model. To facilitate the dialogue with the user all the questions are simply reduced to the confirmation of the changes, or, in case of an error, the choice of the right solution. For example, if the test system reliably determines all the basic controls, it prompts you just to confirm a page layout. Next, the system selects the highest priority operation for testing, then performs it, and analyzes the next state. In case of conflict such as some unexpected behavior or the appearance of the tested application the system will propose to create a characteristic describing the defect.

CONCLUSION

The technology of the test automation using probabilistic networks uses generic templates of interface graphics to conduct the analysis of the interface test program which allows to carry out the testing of the applications based on the "black box" criterion by covering the tolerance range on the basis of the testing criteria of the classes of input and output data.

The developed measures allowed to vary the order of the execution of tests for related modules, analyzing the test results for the current or previous versions of the application and can serve as a new measure to evaluate the relation between the test results and various modules of the program for its overall functionality.

The mechanism of defects detection, designed and tested by the author, can be used to evaluate the correctness of the automated testing work and independently assess the quality of the tested application.

This technology has been tested in the project WebCP by automation Ajax interface testing and has shown its effectiveness and convenience in comparison with the development of GUI Unit Tests writing.

The author thanks his scientific adviser I. Piletski for his help in preparing this paper.

REFERENCES

- [1] G. J. Myers, The Art of Software Testing, John Wiley & Sons, Inc., New Jersey, 2004.
- [2] I. Vinnichenko Automation of testing processes, Peter Press, C-Piterburg, 2005.
- [3] The V-Model as Software Development Standard; IABG Information Technology
- [4] Martin Steinegger and Hannu-Daniel Goiss Introducing a Model-based Automated Test Script Generator - Testing Expirience Magazine. pp.70-75
- [5] S. Russell, P. Norvig, Artificial intelligence: a modern approach (AIMA), Williams, Moscow, 2007.
- [6] R. Shachter, Evaluating influence diagrams. Operations Research, 34 (1986), 871-882.

Software mutation testing: towards combining program and model based techniques

M. Forostyanova
Department of Information technologies
Tomsk State University
Tomsk, Russia
mariafors@mail.ru

N. Kushik
Department of Information technologies
Tomsk State University
Tomsk, Russia
ngkushik@gmail.com

Abstract — The paper is devoted to the mutation testing technique that is widely used when testing different software tools. A short survey of existing methods and tools for mutation testing is presented in the paper. We classify existing methods: some of them rely on injecting bugs into a program under test while other use a formal model of the software in order to inject errors. We also provide a short description of existing tools that support both approaches. We further discuss how these two approaches might be combined for the mutation based test generation with the guaranteed fault coverage.

Keywords — Software testing, mutation testing, mutation operator, model based testing, fault coverage.

I. INTRODUCTION

As the number of widely used information systems increases quickly, the problem of software testing becomes more important. Thorough testing is highly needed for software being used in critical systems such as telecommunications, banking, transportation, etc. [1]. An approach for mutation testing has been proposed around thirty years ago but there still remain some issues of this approach waiting for new effective solutions. Those are fault coverage, equivalent mutants, etc. that we further discuss. In order to solve these problems model based methods for mutation testing are now appearing. In this paper, we make an attempt to follow the chronology of mutation software testing. We start with the initial methodology of mutating programs and further turn to model based mutation testing techniques. In both cases, we provide a brief description of tools which are developed for program/model based mutating testing.

As mentioned in [2], the mutation testing has been introduced by a student Richard Lipton in 1971 [3] while the first publication in this field has been prepared by DeMille, Lipton and Sayward [4]. Meanwhile, the first tool for mutation testing has been developed by Timothy Budd ten years later, in 1980 [5]. Next twenty years the popularity of mutation testing techniques did not grow rapidly while after Millennium it became more and more popular. Moreover, in 2000 the first complete survey of existing methods for mutation testing has appeared [3]. During the last decade there appeared more than 230 publications on mutation testing [2] and almost all existing tools rely on injecting errors in a program under test. One may

turn to [2] to find various papers, PhD theses, etc. combined together into one large repository [6], where the authors make an attempt to cover mutation testing evolution from 1977 till 2009. However, there exist much less publications on model based mutation testing and much less tools that support corresponding formal methods.

In this paper, we first discuss mutation testing technique when a program is mutated by injecting bugs into it. In this case, a program with an injected bug is called a *mutant*. If the behavior of the program is not changed after an injected bug then such injection leads to an *equivalent* mutant. Most of existing tools are developed for software that is written in high level language, and thus, mutation operators are often adapted to language operators. Moreover, ‘good’ tools usually inject those errors that programmer often ignores in his/her programs.

When deriving a mutant based test suite two ways are often used. The first way is to randomly generate test sequences and to check which mutants (errors) are detected by these sequences. Another option is to generate mutant based test sequences such that all the injected errors are detected. The first approach is mostly used when a model is the program itself while the second approach is used more rarely and deals with the formal specification of the program.

Considering the first approach there exist tools that are able to inject bugs into programs written in Fortran [see, for example 7], C/C++ [see, for example 8], Java [see, for example 9], and an SQL code [see, for example 10]. As for the second approach, there exist tools that are developed for injecting errors into software specifications on different abstraction level such as Finite State Machines [see, for example 11], State charts [see, for example 12], Petri Nets [see, for example 13], XML-specification [14]. In this paper, we discuss a number of methods and tools for mutation testing and divide the paper into two parts. The first part is devoted to the *program* based mutation testing where bugs are injected into programs and in the second part the *model* based mutation testing is discussed.

The rest of the paper is organized as follows. Section II contains the preliminaries. Section III is devoted to the program based mutation testing. This section contains the description of a testing method when a program is mutated and a short description of tools for mutating programs written in

Java, C and SQL languages. The model based mutation testing is discussed in Section IV. This approach is illustrated for different kinds of program specifications such as finite state machines, XML-specifications, etc. A number of tools developed for injecting faults into the program specifications are also described. Section V discusses an approach of combining the program based mutation testing with the model based one. Section VI concludes the paper.

II. PRELIMINARIES

As mentioned above, software testing becomes more and more important and there appear new methods and techniques for this kind of testing. Nevertheless, all methods for software testing can be implicitly divided into two large groups. Methods of the first group rely on the informal program specification or the informal software requirements while methods of the second group require a formal model to derive a test suite for a given program. The main advantage of the first approach is the speed of testing that might be rather high because of short length of test sequences and because of a the cardinality of a test suite. However, the main problem of this technique is the fault coverage that is not guaranteed. This problem can be partially solved for model based testing techniques where a test suite is derived based on the formal specification of a given program. This formal specification may be finite transition model [15], pre-post conditions [16], etc. However, the speed of the software testing may fall down exponentially since a long time is needed for deriving formal specifications as well as for deriving a test suite on the basis of this specification. Thus, a good compromise might be to combine somehow methods of the first and the second groups in order to increase the testing speed and to guarantee the fault coverage at least for some classes of program bugs.

Mutant based software testing is not an exception of this tendency and methods for mutation testing can be also implicitly divided into those that rely on a program itself and those that are based on the formal model of a given program. Hereafter, we refer to methods of the first groups as methods of *program based* mutation testing while methods of the second group are called *model based* methods. The main idea of the mutation testing is to change a program or a model in such way that this change corresponds to possible errors in program implementation. Another nontrivial task is to derive a test sequence or a test suite such that all ‘inappropriate’ changes could be detected by applying test sequences.

Each tool for the program based mutation testing relies on a set of mutation operators and this set describes types of errors that can be detected in the source code by a corresponding test suite. The bigger is the set of mutation operators the more test properties can be verified by these mutants.

In this paper, when discussing the program based mutation testing we consider programs written in high level languages like C++ and Java. We further turn to the model based mutation testing and consider finite state machines (FSMs) and extended FSMs (EFSMs) as formal specifications that are widely used for the software test derivation.

Model based testing allows to detect those implementation bugs that cannot be detected by random testing or other techniques of program based testing. Thus, in this paper we discuss which methods and tools are developed for the program based mutation testing as well as for the model based mutation testing. In Section V we make a step towards combining those methods, i.e., we establish a correspondence between software bugs (program mutants) and formal specification errors (model mutants).

III. PROGRAM BASED MUTATION TESTING

In case of the program based mutation testing mutated programs (mutants) are often used for evaluating the quality of a given test suite, i.e., a mutant is used for checking whether corresponding types of program bugs can be detected by the test suite or not. If some mutants cannot be detected or *killed* the test suite is extended by corresponding test sequences. This approach is illustrated in Fig. 1. [2]. One may turn to [2] to find out more about the scheme presented in Fig 1.

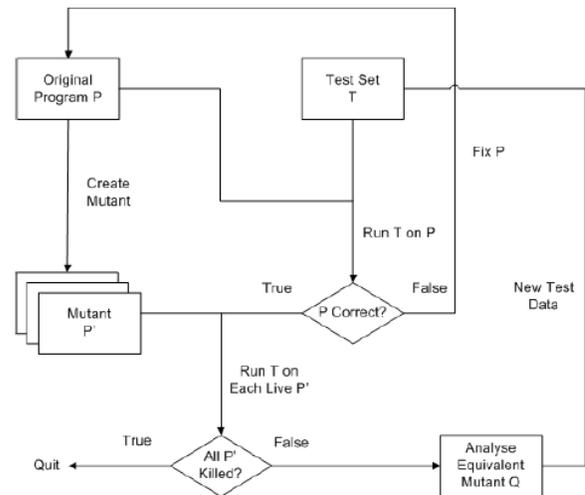


Fig. 1. Generic Process of Mutation Analysis

The program based mutation testing described above has been implemented as several software tools. Most of the tools are developed only for injecting bugs into a source code and only several tools support a test generation process. Moreover, almost every tool for program based mutation testing is commercial. We further provide a short description of existing tools for mutation testing of C/C++ and Java programs.

A. Tools for C program Mutation testing

Agrawal et al. [8] have proposed a comprehensive set of mutation operators for the ANSI C programming language in 1989. There are 77 mutation operators defined in this set. Moreover, Vilela et al. [17] proposed a number of mutation operators to represent bugs associated with static and dynamic memory allocations.

Now there exist a number of tools for injecting errors into C programs. Some of these tools are briefly presented in Table I.

TABLE I

A LIST OF TOOLS FOR PROGRAM BASED MUTATION TESTING FOR C++/C PROGRAMS

Name	Date of first release	Accessibility	Is improving currently	Features
PlexTest [18]	2005	The commercial product	+	Only an instruction removal is supported
Insure++ [19]	1998	The commercial product	+	Injects and detects bugs for identifiers, memory/stack bugs and bugs that concern to linking libraries
Proteum/IM 2.0 [20]	2000	The free download utility	-	Supports 71 mutation operators and calculates the number of mutants being killed
Certitude [21]	2006	The commercial product	+	Can be used for C/ C++ and HDL programs; Combines the mutation approach with the static code analysis; Allows to verify an environment of the program under test
MILU [22]	2008	The free download utility	+	Allows a user to choose the desired number of mutants and to specify their types; 77 mutation operators are supported

One may conclude from Table I that almost all existing tools are developed for injecting bugs, probably except of PlexTest, Insure++ and Certitude that also provide test generation. In spite of the fact that these products are commercial they do not guarantee the guaranteed fault coverage with respect to their specifications.

B. Tools for Java program mutation testing

As many Java programs support the object-oriented paradigm, tools that inject bugs into such programs are mainly concentrated on disturbing inheritance and/or polymorphism features.

Kim et al. [23] were the first to define mutation operators for Java programming language taking into account object-oriented paradigm. This team has proposed 20 mutation operators for Java programs. Moreover, Kim has introduced Class Mutations, which were divided into six groups: Types/Variables, Names, Classes/interface declarations, Blocks, Expressions and others.

Following the tendency from Section A we further describe several tools developed for Java program mutation testing. Differently from C based mutation testing most of the tools developed for the Java program mutation testing are distributed for free. A brief description of available tools is presented in Table II.

Looking at this table one may conclude that there exist tools that support injecting bugs concerned on encapsulation, polymorphism and inheritance. Those are MuJava, Javalanche, and Jester. Moreover, these tools support mutation testing based on corresponding mutation operators. However, the fault coverage of these tests remains unknown. One of the reasons could be the problem of equivalent mutants that are not automatically excluded from the mutants being generated. Therefore, the program based mutation testing needs to be extended with the formal specification of a program under test in order to provide the guaranteed fault coverage of a test suite.

TABLE II

A LIST OF TOOLS FOR PROGRAM BASED MUTATION TESTING FOR JAVA PROGRAMS

Name	Date of first release	Accessibility	Is improving currently	Features
Jester [24]	2001	The free download utility	+	Supports object-oriented mutation operators; Shows equivalent mutants to a user
MuJava [25]	2004	The free download utility	+	Supports 24 mutation operators which specify object-oriented bugs; Mutants are generated and executed automatically; Equivalent mutants have to be excluded manually
MuEclipse [26]	2007	The free download utility (plugin for Eclipse)	+	This is the MuJava version developed for Eclipse
Javalanche [27]	2009	The commercial product	+	Detects around 10% of equivalent mutants; Allows a user to manipulate with a bytecode; Most mutants are concerned about the replacement of an arithmetic operator, constants,

				function calls; Can execute several mutations in parallel and might be used for testing parallel and distributed systems
--	--	--	--	---

IV. MODEL BASED MUTATION TESTING

The first steps in model based mutation testing have been made in 1983 by Gopal and Budd. They have proposed a technique for the software mutation testing describing software requirements taking into account the predicate structure of the program under test.

When generating a test using the model based mutation testing, errors are injected into the model, i.e. the model is mutated. Moreover, similar to the program based mutation testing equivalent mutants need to be deleted. The model based mutation testing has been studied for a number of formal models such as automata models [15], Petri nets, etc described in UML, XML, etc.

By the use of automata models such as FSMs, EFSMs, Petri nets, tree automata, labeled transition systems (LTS), etc. there were proposed a number of approaches for specifying informal software requirements. A number of mutation operators have been proposed for such finite state models. One may turn, for example, to [28] where the authors propose 9 mutation operators representing faults related to states, inputs and outputs of an FSM that is mutated. This set of mutation operators has been implemented in the tool PROTEUM [20]. In [29], the authors investigated an application of mutation testing for probabilistic finite automata (PFAs). They have defined 7 mutation operators and have specified a number of rules how to exclude equivalent mutants.

Mutation operators have been also defined for EFSMs in [30]. In this work, the author has discussed changing operators and/or operands in functions and predicates. Nevertheless, only some types of mutants are formally specified in this work. Thus, in our paper we make an attempt to classify EFSM mutants and to establish a correspondence between EFSM mutants and bugs in the corresponding software implementations.

Tree automata are also of a big help when dealing with software verification. Moreover, each tree automaton can be described as an XML document, and thus, a number of mutation operators is defined especially for XML-documents. One may turn to [14] where Lee and Offut discuss how to inject errors into XML-documents and how to apply this technique for mutation testing of web-servers. The authors have proposed 7 mutation operators and they have further extended their work in 2001 introducing a new approach to XML mutation. This work is based on deriving invalid XML-data using seven mutation operators. All the XML mutation operators introduced in [32] have been combined together and have been implemented in the tool XTM. XTM supports 18 mutation operators and allows to test XML-documents. Nevertheless, the authors of [31] ‘complain’ that only 60% of injected errors have been detected in their experiments.

V. ESTABLISHING A CORRESPONDENCE BETWEEN PROGRAM MUTANTS AND MODEL MUTANTS

In order to somehow combine methods and tools for the program based mutation testing with that based on formal models we are now interested in establishing a correspondence between bugs in a program under test and faults in a model of this program. We are planning to experimentally solve this problem and we focus on testing C/C++ programs. Moreover, we choose one of finite state models discussed above and define a number of mutation operators for this model. A model of an EFSM [32] is rather close to C/C++ implementation because it extends a classical FSM with input and output parameters and context variables. Predicates can be also specified in the EFSM model and a transition can be executed if a corresponding predicate is true. Thus, we establish a correspondence between bugs in C/C++ programs and EFSM faults. Such correspondence can be further used for deriving a test suite for C/C++ implementations based on program mutants but preserving the same fault coverage as if a test suite is derived based on a corresponding EFSM.

We first classify EFSM mutants and then establish to which C/C++ program errors they correspond to.

1. *Predicate EFSM mutant* is derived when a predicate formula is mistaken or the predicate is deleted, i.e. transition becomes unconditional.

2. *Transition EFSM mutant* is derived when a transition is deleted, unspecified transition is added to EFSM or the next state of some transition is wrong.

3. *Function EFSM mutant* occurs when changing a formula for calculating the next value of a context variable or an output parameter.

We now discuss which C/C++ bugs correspond to the above mutants.

A. *Predicate mutants*

Each EFSM predicate corresponds to a *switch/case* or *if/else* instruction of a corresponding C/C++ code, and thus, the following cases are possible.

1. An EFSM predicate is deleted and this fault corresponds to eliminating the *if/else* instruction from the C code.
2. An EFSM predicate consists of several conditions and one of these conditions is deleted. In this case, the corresponding C code contains a complex condition under *if* or *while* and one of its conditions is deleted.
3. Changing logical connectives of a predicate corresponds to a software implementation with an invalid condition.

4. An EFSM predicate can be also changed with respect to a corresponding formula, i.e., operators and/or operands may be changed. These changes correspond to the same changes under *if* or *while* conditions in the C code.

B. Transition mutants

This type of EFSM faults is rather difficult to correlate with C/C++ implementation changes. The reason is that this correspondence strongly depends on how states are defined in the program. If each EFSM state corresponds to one of special program state variable then EFSM transition mutant can correspond to changing the identifier of the next state in the program. If the transition is deleted in the EFSM then a corresponding instruction is deleted from the C/C++ code.

Establishing such correspondence is much more difficult when state semantics is different in the program and by now this option is out of the scope of this paper.

C. Function mutants

When changing formula for calculating values of a context variable or output parameters corresponding C/C++ program is changed in the same way. Thus, EFSM function mutants correspond to those program mutants that are derived by changing corresponding operators and/or operands in the C/C++ instructions.

In Table III the correspondence between EFSM mutants and bugs in software implementations is presented.

TABLE III
A CORRESPONDENCE BETWEEN EFSM MUTANTS AND PROGRAM BUGS

Program bugs	EFSM mutants
Removal of instruction block if / else	Predicate mutant (a transition becomes unconditional)
Removal of a part of composite condition	Predicate mutant
Sign changing in an if condition	Predicate mutant
Operators and/or operands changes in an if condition	Predicate mutant
Change of identifier in an if condition	Predicate mutant
Return of a wrong variable from a function	Output mutant
Changing a sign in an arithmetic operation	Function mutant
Removal of instruction block after if (or else)	Transition mutant

VI. CONCLUSION

In this paper, we have discussed different methods and tools developed for the software mutation testing. The paper clearly shows that there exists a list of tools that support program based mutation testing when a bug is injected into the original program. Much less tools are developed for model based software testing in spite of the fact that this technique allows to guarantee the fault coverage of a test suite. As a result, we are planning to combine the program based mutation testing with the model based one in order to derive tests with the guaranteed fault coverage rather fast. For this purpose we have tried to establish a correspondence between program bugs and model mutants. Such correspondence can be further used for deriving a test suite for C/C++ implementations based on program mutants but preserving the same fault coverage as if a test has been derived based on corresponding EFSM. Developing such testing method based on this correspondence is an open problem for a future work.

REFERENCES

[1] О. Г. Степанов. Методы реализации автоматных объектно-ориентированных программ. Диссертация на соискание ученой степени кандидата технических наук, СПбГУ ИТМО: 2009, 115 с.

[2] Yue Jia, M. Harman, IEEE, "An Analysis and Survey of the Development of Mutation Testing", pp 33

[3] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the Orthogonal," In Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00), published in book form, as Mutation Testing for the New Century. San Jose, California, 6-7 October 2001, pp. 34-44

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol. 11, no. 4, pp. 34-41, April 1978

[5] T. A. Budd, "Mutation Analysis of Program Test Data," PhD Thesis, Yale University, New Haven, Connecticut, 1980

[6] Repository: [web-site]. URL: <http://www.dcs.kcl.ac.uk/pg/jiayue/repository/>, 2010

[7] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs," in Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'80), Las Vegas, Nevada, 28-30 January 1980, pp. 220-233

[8] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-41-P, March 1989.

- [9] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," in Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00), published in book form, as Mutation Testing for the New Century. San Jose, California, 6-7 October 2001, pp. 207–225.
- [10] SQLmutation : [web-site]. URL: <http://in2test.lsi.uniovi.es/sqlmutation/>, 2005
- [11] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar, "Specification of Timed EFSM Fault Models in SDL," in Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'07), ser. LNCS, vol. 4574. Tallinn, Estonia: Springer, 26-29 June 2007, pp. 50–65.
- [12] G. Fraser and F. Wotawa, "Mutant Minimization for Model-Checker Based Test-Case Generation," in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07), Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 161–168.
- [13] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and W. E. Wong, "Mutation Testing Applied to Validate Specifications Based on Petri Nets," in Proceedings of the IFIP TC6 8th International Conference on Formal Description Techniques VIII, vol. 43, 1995, pp. 329–337.
- [14] S. C. Lee and A. J. Offutt, "Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis," in Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01), Hong Kong, China, November 2001, pp. 200–209.
- [15] N. Shabdina, Khaled El-Fakih, N. , "Testing Nondeterministic Finite State Machines with Respect to the Separability Relation", TestCom/FATES, 2007, pp:305-318
- [16] S. S. Batth, E. R. Vieira, A. R. Cavalli, and M. U. Uyar, "Specification of Timed EFSM Fault Models in SDL," in Proceedings of the 27th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'07), ser. LNCS, vol. 4574. Tallinn, Estonia: Springer, 26-29 June 2007, pp. 50–65.
- [17] P. Vilela, M. Machado, and W. E. Wong, "Testing for Security Vulnerabilities in Software," in Software Engineering and Applications, 2002.
- [18] PlexTest : [web-site]. URL: <http://www.itregister.com.au/products/plextest>, 2005
- [19] Insure++: [web-site]. URL: <http://www.parasoft.com/jsp/products/insure.jsp?itemId=63>, 1998
- [20] M. E. Delamaro, J. C. Maldonado, "Proteum/IM 2.0: An Integrated Mutation Testing Environment", Univ. de Sao Paulo, Sao Paulo, Brazil, 2001, pp. 91 - 101
- [21] Certess, "Certitude," : [web-site]. URL: <http://www.certess.com/product/>, 2006
- [22] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08). Windsor, UK: IEEE Computer Society, 29-31 August 2008, pp. 94–98.
- [23] S. Kim, J. A. Clark, and J. A. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP," in Proceedings of the 12th International Conference Software and Systems Engineering and their Applications (ICSSEA '99), Paris, France, 29 November-1 December 1999.
- [24] Jester : [web-site]. URL: <http://jester.sourceforge.net/>, 2001
- [25] MuJava : [web-site]. URL: <http://cs.gmu.edu/~offutt/mujava/>, 2004
- [26] B. H. Smith and L. Williams, "An Empirical Evaluation of the MuJava Mutation Operators," in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07). Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 193–202.
- [27] Schuler, D., Dallmeier, V., and Zeller, A. 2009. Efficient mutation testing by checking invariant violations. In Proceedings of the 18th international Symposium on Software testing and Analysis (2009), pp. 69–80.
- [28] S. P. F. Fabbri, M. E. Delamaro, J. C. Maldonado, and P. Masiero, "Mutation Analysis Testing for Finite State Machines," in Proceedings of the 5th International Symposium on Software Reliability Engineering, Monterey, California, 6-9 November 1994, pp. 220–229.
- [29] R. M. Hierons and M. G. Merayo, "Mutation Testing from Probabilistic Finite State Machines," in Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07), published with Proceedings of the 2nd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'07). Windsor, UK: IEEE Computer Society, 10-14 September 2007, pp. 141–150.
- [30] A. В. Коломеец. Алгоритмы синтеза проверяющих тестов для управляющих систем на основе расширенных автоматов. Дис. ... канд. техн. наук. Томск: 2010, 129 с.
- [31] Ledyvania Franzotte and Silvia Regina Vergilio, "Applying Mutation Testing to XML Schemas", Computer Science Department, Federal University of Parana (UFPR), Brazil, pp 6
- [32] El-Fakih K., Prokopenko S., Yevtushenko N., Bochmann G. "Fault diagnosis in extended finite state machines", In Proc. of the IFIP, 15th Intern. Conf. on Testing of Communicating Systems, France -2003.

Experimental comparison of the quality of TFSM-based test suites for the UML diagrams

Rustam Galimullin

Department of Radiophysics
Tomsk State University
Tomsk, Russia
nihilkhaos@gmail.com

Abstract— The paper presents the experimental comparison of the quality of three test suites based on the model of a Finite State Machine with timeouts, namely, the explicit enumeration of faulty mutants, transition tour and TFSM-based black-box test. Test suites are then applied to the program, automatically generated via the UML tool. The experimental results on the quality of the above mentioned test suites and the corresponding analysis are presented.

Keywords—Finite State Machines with timeouts, the UML state machine diagrams, test suites

I. INTRODUCTION

Nowadays software failures of critical control systems are very expensive, and, thus, it is essential to provide high-quality testing at every stage of the system development. Many of such systems are formally described using the UML (the Unified Modeling Language) that has become the *de facto* standard for modeling software applications. The UML being a visual modeling language allows obtaining comprehensive and detailed information about a system under design, as well as provides a possibility for convenient update of the system. Correspondingly, the UML is widely used in software engineering, business project development, hardware design and in a number of other applications. The UML description can be automatically translated into a program code using proper tools and the developed software should be thoroughly tested. One of the formal models for testing UML-based software is a trace timed model. In this paper, we derive tests with the guaranteed fault coverage based on a timed Finite State Machine (FSM) augmented with timeouts [1], since FSMs are known to be an efficient model for deriving tests with the guaranteed fault coverage. The paper presents a case study for assessing the quality of test suites derived by three methods [2, 3, 4], which is estimated for the example of a phone line; the UML description of this project is taken from [5]. Using the tool *Visual Paradigm for UML* 8.0 [6] a JAVA code is generated for this application that serves a sample when assessing the test suite quality. We first check whether the initial program passes all the derived test suites. At the second step, some practical faults are injected into the initial program. Applying to each mutant tests, which were derived on the basis of timed FSM, we check whether injected faults

can be detected with the test suites and analyze the reason when some faults cannot be detected with some/all derived test suites.

II. PRELIMINARIES

The model we use, TFSM, is an extension of a classical FSM that is described as a finite set of states and transitions between them. Every transition is labeled by an *input/output* pair, where an input triggers the transition and an output is a system response to a given input. Formally, a *timed Finite State Machine* (TFSM) is a 6-tuple $S = (S, I, O, s_0, \lambda_S, \Delta_S)$ where S is a finite nonempty set of states with the initial state s_0 , I and O are finite disjoint input and output alphabets, $\lambda_S \subseteq S \times I \times O \times S$ is transition relation and $\Delta_S: S \rightarrow S \times (\mathbb{N} \cup \infty)$ is a delay function defining timeout for each state [1]. If no input is applied at a current state during the appropriate time period (timeout), a TFSM can move to another prescribed state. A TFSM is called *deterministic* if for each pair $(s, i) \in S \times I$ there is at most one pair $(o, s') \in O \times S$ such that $(s, i, o, s') \in \lambda_S$, otherwise it is called *nondeterministic*. If for each pair $(s, i) \in S \times I$, there is at least one pair $(o, s') \in O \times S$ such that $(s, i, o, s') \in \lambda_S$ then S is said to be *complete*, otherwise it is *partial*.

A *timed input* symbol is a pair $\langle i, t \rangle \in I \times \mathbb{Z}_0^+$, where \mathbb{Z}_0^+ is a set of nonnegative integers. The timed input symbol shows that the input symbol i is applied at the moment when the value of the time variable is equal to t . A sequence of timed input symbols $\langle i_1, t_1 \rangle \dots \langle i_k, t_k \rangle$ is a *timed input sequence* of length k .

Let $S = (S, I, O, s_0, \lambda_S, \Delta_S)$ and $Q = (Q, I, O, q_0, \lambda_Q, \Delta_Q)$ be complete TFSMs. TFSMs S and Q are said to be *non-separable* if the sets of output responses of these TFSMs to any timed input sequence α intersect; i.e. $out_S(s_0, \alpha) \cap out_Q(q_0, \alpha) \neq \emptyset$. Otherwise, the TFSMs are *separable*. A timed input sequence α , such that $out_S(s_0, \alpha) \cap out_Q(q_0, \alpha) = \emptyset$ is called a *separating sequence* for TFSMs S and B . TFSM S is a *submachine* of TFSM Q if $S \subseteq Q$, $s_0 = q_0$ and each timed transition $(s, \langle i, t \rangle, o, s')$ of S is a timed transition of Q .

Intersection $S \cap Q$ of two FSMs is the largest submachine of $P = (P, I, O, p_0, \lambda_P, \Delta_P)$, where $P = S \times K \times Q \times K$, $K = \{0, \dots, k\}$, $k = \min(\max \Delta_S(s), \max \Delta_Q(q))$, the initial state is

quadruple $(s_0, 0, p_0, 0)$. Transition relation λ_P and function Δ_P are defined by the following rules:

1. The transition relation λ_P contains quadruple $[(s, k_1, q, k_2), i, o, (s', 0, q', 0)]$ iff $(s, i, o, s') \in \lambda_S$ and $(q, i, o, q') \in \lambda_Q$.
2. Time function is defined as $\Delta_P(s, k_1, q, k_2) = [(s, k_1', q, k_2'), k]$, $k = \min(\Delta_S(s)_{\downarrow N} - k_1, \Delta_Q(q)_{\downarrow N} - k_2)$. State $(s', k_1', q', k_2') = (\Delta_S(s)_{\downarrow S}, 0, \Delta_Q(q)_{\downarrow Q}, 0)$, if $\Delta_S(s)_{\downarrow N} = \infty$, $\Delta_Q(q)_{\downarrow N} = \infty$ or $(\Delta_S(s)_{\downarrow N} - k_1) = (\Delta_Q(q)_{\downarrow N} - k_2)$. If $(\Delta_S(s)_{\downarrow N} - k_1), (\Delta_Q(q)_{\downarrow N} - k_2) \in \mathbb{Z}_+$ и $(\Delta_S(s)_{\downarrow N} - k_1) < (\Delta_Q(q)_{\downarrow N} - k_2)$, then state $(s', k_1', q', k_2') = (\Delta_S(s)_{\downarrow S}, 0, q, k_2 + k)$. If $(\Delta_S(s)_{\downarrow N} - k_1), (\Delta_Q(q)_{\downarrow N} - k_2) \in \mathbb{Z}_+$ and $(\Delta_S(s)_{\downarrow N} - k_1) > (\Delta_Q(q)_{\downarrow N} - k_2)$, then state $(s', k_1', q', k_2') = (s, k_1 + k, \Delta_Q(q)_{\downarrow Q}, 0)$.

III. CASE STUDY

As a running example, we consider a simple phone line state machine diagram, taken from [5].

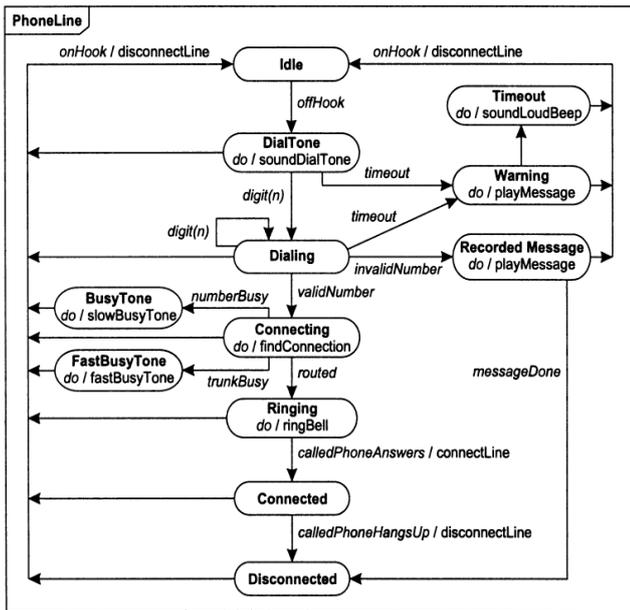


Fig. 1. Phone line state machine diagram

When the device is at *Idle* state it is possible to pick up the phone (*offHook*), and to get *soundDialTone* as an output. The state diagram is at *Dialing* state when a user enters the number (*digit(n)*). If the number cannot be served (*invalidNumber*), the corresponding message is played (*playMessage*). Otherwise, the device enters the state *Connecting*. At this state, four different events are possible. Either the number or a trunk is busy, and in this case, the user should hang up, or the phone will connect (*routed*). After the connection there is the ring (*ringBell*) and, finally, the conversation takes place (state *Connected*). After the conversation, either the user or her/his partner hangs up. In both cases, the line will be disconnected. With a case tool *Visual Paradigm for UML 8.0 JAVA* program code of this diagram was automatically generated.

A TFSM that describes such a state machine diagram is in Figure 2.

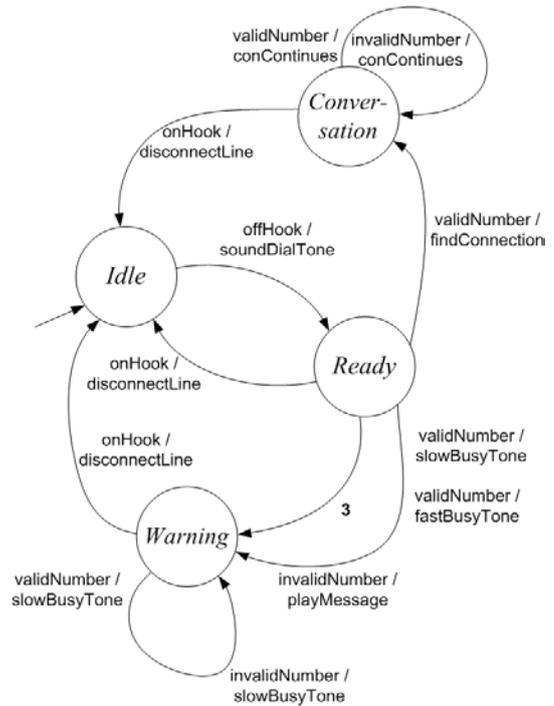


Fig. 2. TFSM that describes the phone line, presented in Fig. 1.

The TFSM has four states and one timeout at state *Ready*. The initial state is *Idle*. If the user picks up the phone (*offHook*), a dial tone is played (*soundDialTone*), and the TFSM changes its state to *Ready*. If the user does not interact with the system for a certain period of time, 3 time units for instance, the state will be spontaneously changed to *Warning*. At the *Ready* state, a user also can hang up the phone (*onHook*) and in this case, the line will be disconnected (*disconnectLine*). If the user enters a valid number (*validNumber*), the TFSM can response in three different ways. The first option is a busy number (*slowBusyTone*) and in this case, the system changes its state to *Warning*. The second option is a busy trunk (*fastBusyTone*). The last option is that a conversation starts. In other words, the corresponding TFSM is nondeterministic. In the *Warning* state none of entered numbers (*validNumber* and *invalidNumber*) affects the system. Being in this state the user can only hang up (*onHook*) and the same situation occurs in the *Conversation* state.

Here we notice that a TFSM is also partial and cannot be augmented to a complete TFSM as it is usually done when deriving tests against a partial specification FSM. The reason is that after *onHook* input we cannot apply the same input. This input can be applied only after the input *offHook*.

IV. METHODS OF TEST DERIVATION

Three TFSM-based methods for the test suite derivation are considered in the paper. The first method (Method 1) is based on the explicit enumeration of faulty mutants. Given the specification TFSM, some faults are injected in it, i.e. some mutant TFSMs are constructed, and for each mutant an input sequence that separates the specification TFSM and this mutant is derived [2]. A separating sequence is an input sequence such that the sets of output responses of two TFSMs to this sequence do not intersect, and since the TFSMs can be nondeterministic we use a separating sequence instead of traditional distinguishing sequence [7]. In order to derive a separating sequence we first construct the intersection of two given TFSMs and then a truncated successor tree is

constructed for the intersection. In the paper, we consider only six mutants which describe meaningful faults for our running example.

1. A fault related to the timeout at state *Ready*. In this case, the TFSM has a transition, labelled with timeout 4, instead of 3. For this pair of FSMs, specification (Fig. 2.) and mutant TFSMs, a separating sequence is $(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 3 \rangle)$.
2. Another wrong timeout. But now it is smaller (e.g. 1) than that of the specification TFSM. By direct inspection, one can assure, that for this mutant a separating sequence is $(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 2 \rangle)$.
3. The situation when having an invalid number as an input the connection is still found. For this particular case, a separating sequence is $(\langle \text{offHook}, 0 \rangle, \langle \text{invalidNumber}, 0 \rangle)$.
4. The situation when during the conversation a user accidentally types some digits (a number), and the slow busy tone is played. In this case, an input sequence $(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 0 \rangle, \langle \text{validNumber}, 0 \rangle)$ is a separating sequence.
5. The situation when being at state “Warning” we can make a call anyway. For this case, a separating sequence is $(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 3 \rangle)$.
6. The situation when conversation is impossible (i.e. there is no transition to state *Conversation*) and in this case, a separating sequence is $(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 0 \rangle)$.

We consider the set of the above mentioned separating sequences as a test suite for explicit enumeration of mutants. Thus, $TS_1 = \{(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 3 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 2 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{invalidNumber}, 0 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 0 \rangle, \langle \text{validNumber}, 0 \rangle)\}$.

The second method (Method 2) for deriving a test suite against TFSMs with the guaranteed fault coverage is based on the correlation between TFSM and FSM (Procedure 1) [4]. To transform a timed FSM into a classical FSM we add a special input symbol 1 that corresponds to the notion of *waiting one time unit*, and a special output – *N* that corresponds to the case when there is no reply from the machine. If at state *s* a timeout value *T* is greater than 1, then we add $(T - 1)$ copies of state *s* with corresponding outgoing transitions. If TFSM have *n* states and the maximum finite timeout is T_{max} , the corresponding FSM may have up to $n \cdot T_{max}$ states. In Figure 3, there is an FSM that corresponds to the specification TFSM in Figure 2.

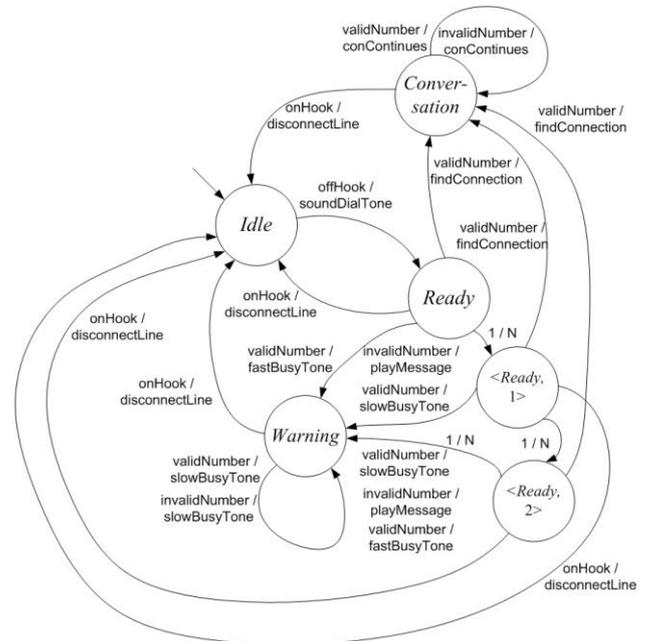


Fig. 3. FSM that corresponds to the TFSM, presented in Fig. 2.

Given a classical FSM, a test suite that is complete w.r.t. to output faults can be derived as a transition tour of the FSM [3]. A transition tour of an FSM is a finite set of input sequences which started at the initial state traverse each FSM transition. A corresponding transition tour can be derived for an FSM that is derived from corresponding TFSM.

Proposition 1. Given a test suite TS for TFSM based on a transition tour of an FSM output by Procedure 1, the TS detects each output fault of the TFSM.

A transition tour for the FSM (Fig. 3) is a test suite $TS_2 = \{(\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 1 \rangle, \langle \text{validNumber}, 0 \rangle, \langle \text{onHook}, 0 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 1 \rangle, \langle \text{validNumber}, 0 \rangle, \langle \text{onHook}, 0 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 0 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{invalidNumber}, 2 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{validNumber}, 2 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{onHook}, 1 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{offHook}, 2 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{onHook}, 0 \rangle), (\langle \text{offHook}, 0 \rangle, \langle \text{invalidNumber}, 0 \rangle)\}$.

Consider now the third test derivation method proposed in the paper [4]. The method has two testing assumptions: the upper bound on the number of states of a TFSM under test (implementation under test, IUT) and the largest finite timeout at a state of the IUT are known. Authors show that in this case, a complete test suite obtained directly from a given TFSM is much shorter than a complete test suite that is derived based on a corresponding FSM by the use of corresponding FSM based methods [8]. The procedure for test derivation consists of three steps. We first identify each state of the specification TFSM using separating sequences. At the next step, we check all transitions at each state, i.e. reach a state, execute a transition and execute corresponding separating sequences. At the last step, timeouts are tested: for this purpose at each state we apply inputs $(i, 1), \dots, (i, T + 1)$ when T is the largest timeout of the IUT. The method was proposed for reduced complete deterministic TFSMs; however, we use it also for nondeterministic partial TFSMs adding separating sequences after each transition. We also assume that if a timeout at a state of the specification TFSM is ∞ then the IUT has the same timeout. Correspondingly, for the specification TFSM (Fig. 2.) we do not check the initial state *Idle*; all other states can be identified by separating sequences listed below. For

state *Warning* we have a separating sequence $\langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle\rangle$, for state *Conversation* - $\langle\langle\text{offHook}, 0\rangle, \langle\text{validNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle\rangle$ and for state *Ready* - $\langle\langle\text{offHook}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle$. At the second step all the transitions are checked. We use a transition tour $\{\langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{validNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle\}$, where each sequence is augmented with a corresponding separating sequence. At the final step timeouts are checked and we derive the following sequences: $\{\langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 1\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 2\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 3\rangle\rangle\}$. Thus for given FSM test suite is $TS_3 = \{\langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{validNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{validNumber}, 0\rangle, \langle\text{invalidNumber}, 0\rangle, \langle\text{onHook}, 0\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 1\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 2\rangle\rangle, \langle\langle\text{offHook}, 0\rangle, \langle\text{invalidNumber}, 3\rangle\rangle\}$.

V. EXPERIMENTAL RESULTS

We now consider the set of possible program faults which is listed below.

1. The transition from state *Ready* to state *Conversation* is triggered by input *validNumber*, but the output is *fastBusyTone* instead of *findConnection*.
2. The timeout at state *Ready* is greater than that in the specification TFSM, e.g. timeout equals six instead of three.
3. There is a new transition from state *Ready* to state *Warning* under input *onHook* with *fastBusyTone* output.
4. A fault is inside the program. While scanning the valid number set there is a *while loop*, and if an input number coincides with one in the list, then Boolean variable *flag* is true, otherwise – false.

```
while ((strLine = br.readLine()) != null)
{
    if (s == null ? strLine == null :
        s.equals(strLine))
    {
        flag = true;
    }
}
```

The fault is as follows. If an entered number is *not* in the list, then *flag* is still true and in order to get the mutant we add **!** in an *if clause*.

```
while ((strLine = br.readLine()) != null)
{
    if (s == null ? strLine == null :
        !s.equals(strLine))
    {
        flag = true;
    }
}
```

The fault implies that all entered numbers are valid.

5. A new state is added. From state *Ready* it is possible to enter a new *Wait* state via *validNumber* input. This means that having a

valid number as an input a user would listen to a special message, e.g. “Connection is set up. Please wait”. This is modeled by an output *findConnection*. On input *validNumber* in state *Wait* there is an output *convContinues*. If an *invalidNumber* is entered the implementation TFSM changes its state to state *Warning* with *fastBusyTone* output. Finally, if *onHook* input is applied the machine is at *Idle* state and *disconnectLine* output is produced.

6. There is a new timed transition from state *Conversation* to state *Ready* after 8 time units. This means, that after 8 time units the conversation is automatically finished.

We first apply each test case to the initial program to be sure that the program produces expected output sequences to every test case. Then all the above faults were injected into the initial program. For each test suite, each test case was applied to a mutant program. A fault was *detected* by a test suite when there was at least one test case of the test suite such that the output responses of the initial program and of a mutant program were different. The results are presented in Table I, where ‘+’ means that this fault can be detected by a corresponding test suite.

TABLE I. EXPERIMENTAL RESULTS

	1	2	3	4	5	6
TS_1	+	+	-	+	-	-
TS_2	+	+	+	+	-	-
TS_3	+	+	+	+	+	-

As we can see, Faults 3, 5 and 6 were not found by TS_1 ; the reason can be that we did not consider corresponding mutants for the specification TFSM. Despite of the fact, that some of such mutant program still can be detected in this case, we were ‘unlucky’. Test suite TS_2 did not detect Faults 5 and 6, since when considering a transition tour, we assume that the number of states of an IUT is the same as of the specification TFSM. Finally, Fault 6 was not detected even by TS_3 because we also violated testing hypothesis about an implementation TFSM. Nevertheless, we could conclude that a transition tour where each sequence is appended with corresponding separating sequence can detect more faults and thus, such augmentation is worth for improving the quality of a generated test suite.

V. CONCLUSIONS

In this paper, we considered three methods for the deriving tests based on the model of an FSM augmented with input and output timeouts for automatically generated program code of an UML project. Using a simple running example we illustrate that a transition tour of the specification TFSM augmented with corresponding separating sequences is a test suite of a good quality and this test suite detects not only faults it is derived for, but also other faults, including those which increase the number of states of an implementation TFSM.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my scientific supervisor professor Nina Yevtushenko for her invaluable support during the work on paper.

REFERENCES

- [1] M. Gromov, D. Popov and N. Yevtushenko, "Deriving test suites for timed Finite State Machines," Proc. of IEEE East-West Design & Test Symposium, pp. 339-343, 2008.
- [2] N. Shabaldina and R. Galimullin, "On deriving test suites for nondeterministic Finite State Machines with time-outs," Programming and computer science, vol. 38, pp. 127-133, 2012.
- [3] M. Zhigulin "TFSM-based methods of fault detection tests synthesis with guaranteed fault coverage for discrete controlling systems", PhD thesis, TSU, Tomsk, 2012. (in Russian)
- [4] M. Zhigulin, S. Maag, A. Cavalli and N. Yevtushenko, "FSM-based test derivation strategies for systems with time-outs," Proc. of the 11th conference on quality software (QSIC), pp. 141-149, 2011.
- [5] J. E. Rumbaugh and M.R. Blaha "Object-oriented modeling and design with UML (2 ed.)," Pearson Education, 2005.
- [6] Visual Paradigm [Electronic resource] - <http://www.visual-paradigm.com/>
- [7] A. Gill, "Introduction to theory of Finite State Machines," McGraw-Hill, 1962.
- [8] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, N. Yevtushenko (2010) "FSM-based Conformance Testing Methods: A Survey Annotated with Experimental Evaluation," Information and Software Technology, Elsevier, 52, pp. 1286-1297, 2010.

Experience of Building and Deployment Debian on Elbrus Architecture

Andrey Kuyan, Sergey Gusev, Andrey Kozlov, Zhanibek Kaimuldenov, Evgeny Kravtsunov
Moscow Center of SPARC Technologies (ZAO MCST)
Vavilova street, 24, Moscow, Russia
{kuyan_a, gusev_s, kozlov_a, kajmul_a, kravtsunov_e}@mcst.ru

Abstract—This article describe experience of porting Debian Linux distribution on Elbrus architecture. Authors suggested effective method of building Debian distribution for architecture which is not supported by community.

I. INTRODUCTION

MCST (ZAO "MCST") is a Russian company specializing in the development of general purpose CPU with Elbrus-2000 (e2k) ISA [1] and computing platforms based on it [2]. Also in the company are being developed optimizing and binary compilers, operating systems. General purpose of microprocessors and platforms assume that users have the ability to solve any problems of system integration with its help. At the user level universality is provided by distribution of operating system. Distribution uses architecture-dependent capabilities of software components, such as the kernel, architecture-dependent system libraries and utilities, compilers. Nowadays there are several large and widely used distributions supported by community: Gentoo, Slackware, Debian. We chose Debian guided by wishes of our customers and because Debian is one of the most stable and well supported Linux distributions. Debian has system of package management, installer and all this components are supported by a large community of developers. 99% of Debian packages are architecture independent applications and libraries, which is written in C/C++ or Perl, Python, etc. There is a popular way to building distribution for unsupported by community architecture: download a limited number of software sources with different versions and add a popular package manager, for example dpkg with limited functionality. This approach allows to provide for user a distribution with basic functionality and even call it a Debian-like. A significant drawback of this way is the complexity or even impossibility of extending the package set. Due to software dependencies, even if they are, do not match with dependencies of pure Debian and nothing additional can be built with using dpkg-buildpackage. This drawback is not so significant for specialized systems that solve the limited range of tasks, but a problem for the platform, which is claimed as universal. This problem is solved by porting Debian on new architecture in its purest form with preserving all dependencies. The resulting distribution will allow to solve all kind of the current problems, even more a system integrator will be able to build new packages using the package manager.

II. DEBIAN PACKAGE MANAGEMENT SYSTEM

Debian is built from a large number of open-source projects which maintained by different groups of developers around the world. Debian uses the package term. There are 2 types of packages: source and binary. Common source package consists of *.orig.tar.gz file, *.diff.gz file and *.dsc file. *.orig.tar.gz file contains upstream code of a project, maintained by original developers. *.diff.gz file contains a Debian patch with some information about project, such as build-dependencies, build rules, etc. *.dsc file holds an information about *.orig.tar.gz and *.diff.gz. Some source packages, maintained by Debian developers (for example dpkg) may not comprise *.diff.gz file because they already have a Debian information inside. Binary packages can contain binary and configuration files, scripts, man pages, documentation and another files to install on the system. In addition, each package holds metadata about itself. Binary packages represented as *.deb files. Source and binary packages contains information about build and runtime dependencies respectively. Build dependencies are binary packages that has to be installed on the system for building source package. Runtime dependencies are binary packages that has to be installed on the system for correct work of package.

In fact, Debian solves two main problems:

- 1) supporting appropriate versions of packages
- 2) managing packages with dpkg and support tools

For building any source package, some utilities have to be installed on the system. For example, every source code required make utility. Set of packages that have to be installed on the system for building called build-essentials. Some of them are Debian-specific. This because Debian has it's own tools and features for building source code directly into packages. Debian patch for source package holds, as mentioned, build rules. Rules is the makefile with set of standard targets, such as "clean" and "build". Build process starts with run dpkg-buildpackage script which is part of dpkg-dev package. This script checks build dependencies, gets some information about environment and runs the desired targets from rules.

III. ARCHITECTURE-DEPENDENT SOFTWARE AND DEBIAN RELEASE SELECTION

Architecture-dependent part of software stack (see Fig.1) comprises the following components: Linux kernel, glibc

library, toolchain (compiler lcc and binutils), strace and debugger gdb. Development of this components for new CPU architecture is long and laborious process. Versions of this components are crucial for Debian release number selection.

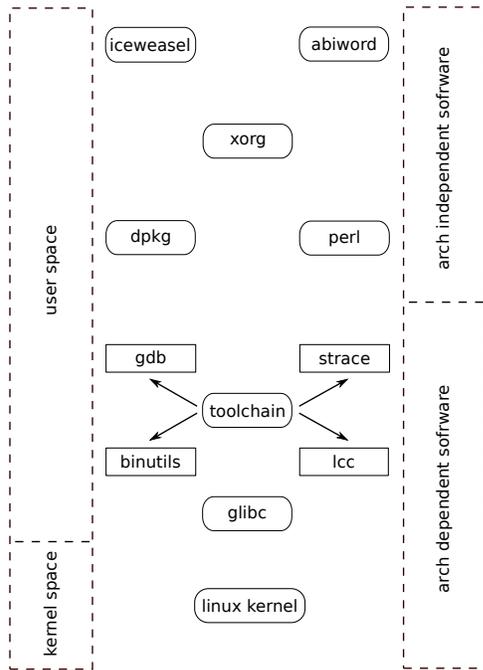


Fig. 1. Software stack

In the table 1 represented comparison of MCST and Debian architecture-dependent components, according to this table Debian Lenny is appropriate one for porting.

TABLE I
ARCHITECTURE-DEPENDENT COMPONENTS COMPARISON

version	kernel	glibc	binutils	gcc
MCST	2.6.33	2.7	2.18	3.4.6
Lenny	2.6.26	2.7	2.18	4.3.2
Squeeze	2.6.32	2.11	2.20	4.4.5
Wheezy	3.2	2.18	2.10	4.7.2

MCST compiler team has developed two kinds of toolchain: cross and native. Cross-toolchain, which is running x86-machine, allow to generate code for e2k ISA. MCST architecture-dependent components consist of binutils-2.18, glibc-2.7, gdb-7.2, gcov, dprof, libstlport, libffi and lcc (compiler compatible with gcc 3.4.6). Compiler lcc is original development of MCST [3] and uses frontend which is licensed from Edison Design Group (EDG)[4]. Remaining components are the result of porting the GNU utilities on e2k-architecture and contain a large number of changes arising from architectures peculiarities.

IV. TECHNICAL ISSUES FOR CHALLENGING

Define main technical problems in porting Debian on a new CPU architecture:

- 1) The chicken and egg problem: for build any package we need build-essentials set. But we haven't this one because we haven't build and runtime dependencies for build-essentials packages.
- 2) Some packages may cyclically dependent on each other. Example of cyclic dependencies shown in figure 2.
- 3) While building build-essentials there isn't simple way to pass arch identifier to configure script. This because we built build-essentials without dpkg, which provides features for auto-detecting cpu type. The problem is compounded by the fact that some packages strictly depends on the architecture type.
- 4) Compilation speed problem. For running iceweasel, gnumeric and so on we need to have almost 2500 binary packages in our repository. Some of them are very large and build takes more than a day.
- 5) Difference between gcc and elbrus toolchains: gcc toolchain packages set and elbrus toolchain packages set vary greatly, elbrus compiler don't support some new language extensions or compiler flags.

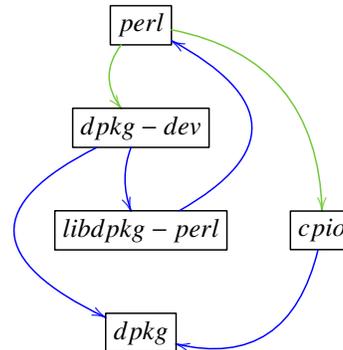


Fig. 2. Example of cyclic dependencies.

Fig.2 illustrates a part of runtime and build dependencies graph for dpkg package. This graph represents existence of cyclic dependencies, blue arrows depict runtime dependencies and green arrows depict build dependencies. Package dpkg-dev is build dependence of perl and it is used only in process of building perl package, but dpkg-dev required for working package libdpkg-perl, which depend on perl. Thus perl interpreter should run on machine for building perl package. Solutions of these problems are presented below.

V. SOLUTION FOR PACKAGE MANAGER: BUILDING ESSENTIALS, BREAKING CYCLIC DEPENDENCIES

Packages from build-essentials set have been built in the following algorithm:

- 1) Using debtree utility we built dependency graph of required packages for build-essentials set.
- 2) Every package from graph has been built with native toolchain and configure-make mechanism, without dpkg.
- 3) In build process, we broke some cyclic dependencies. Break dependencies algorithm shown in figure 2. It

is seen that if package A depends on package B and package B depends on package A, we should build package B 2 times: first time with broken A dependence, which mean we don't pass corresponding option to configure second time after building A package in due form.

- 4) Result of building has been installed on the machine and at the same time wrapped in the package manually.
- 5) After building all graph elements we built dpkg with configure-make mechanism.
- 6) Then we verified package manager efficiency by install on the machine all deb packages that we got manually.
- 7) All build-essential set have been rebuild with dpkg-buildpackage.

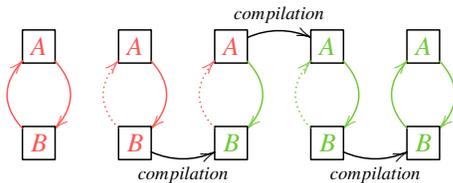


Fig. 3. Algorithm of resolving circular dependencies

After sequential implementation all 7 points of algorithm we've got small system which can be used for building all other necessary packages with dpkg. So, manual building with configure-make used only in initial phase. After dpkg all packages have been building with standard debian rules.

VI. SOLUTION FOR COMPILER PROBLEMS

As was mentioned above, Debian for e2k ISA is based on using lcc compiler, which was developed by MCST compiler team. Lcc compiler is using edg frontend, which is compatible with gcc 3.4.6. Lcc doesn't support some extensions of C language, for example nested functions. Programs written with using of nested functions, should be patched to unwrap this functions. Fortunately as the experience of porting Debian distribution nested functions are seldom used, so patches require only for several packages. One of those packages is bogl-bterm, which is used by Debian Installer as a graphical frontend. Many of software developed by GNU project are using gcc directives `__attribute__((attribute-list))`, and some of this directives are not supported by lcc, so for succesful compilation of such code corresponding patches should be applied to source package. MCST toolchain contain library for STL support - libstlport, which is different from standard STL library libstdc++, also libstlport is not supported some STL features. Due to this distinctions such packages as mysql and exim require patches or special tuning in makefiles to be successfully compiled.

VII. HYBRID COMPILE FARM

Due to the existence of two toolchains (cross and native) feasibility to decrease compilation time appeared. This technique is based on using hybrid scheme of source-code com-

pilation via distcc. Distcc (distributed C/C++/ObjC compiler)

TABLE II
COMPILE FARM CONFIGURATION

parametr	e2k	x86
CPU name	Elbrus-2C+	Core2 Duo E8400
CPU frequency	500 MHz	3.00GHz
Number of cores	2 + 4(dsp)	2

[5] is a software for speeding up compilation process by using distributed computing over network. Compilation of source package is starting on e2k host with native toolchain and distcc client, this client is sending preprocessed files to servers with x86-architecture for code compilation using cross-toolchain. After compilation server return object files to clients which perform operations of linking and *.deb packages forming. Hybrid compile farm, which configuration is described in Tab. 2, was used for building linux distribution of 350 source packages. This method allow to decrease up to 5 times average time of package building as compared with native compilation. Fig. 2 shows a generalized scheme of compile farm, which is described above.

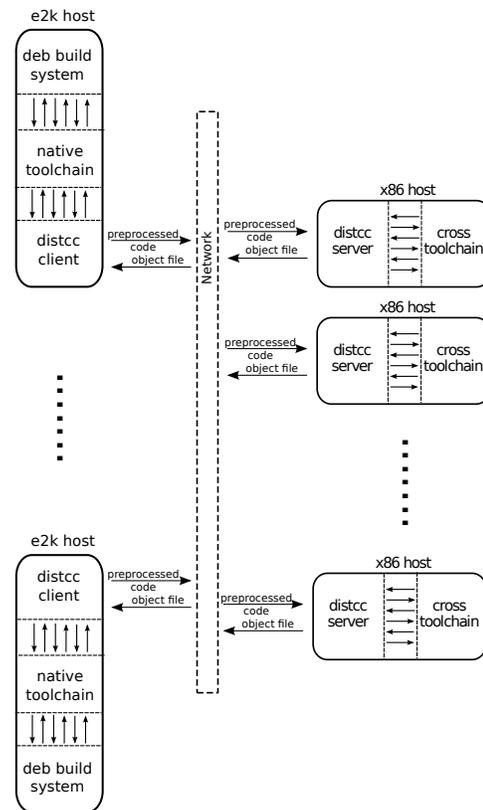


Fig. 4. Hybrid compile farm

VIII. SOLUTION FOR ARCH TYPE

When any package is builded with configure-make, there are two main ways to pass arch identifier to the configure script: pass `--build=arch` option to the configure fix

config.guess script which contains all known by community arch identifiers For correct building we used both. Typical config.guess patch is as follows:

```
--- config.guess-orig
+++ config.guess-fix
@@ -854,6 +854,9 @@
     crisv32:Linux:*:*)
         echo crisv32-axis-linux-gnu
         exit ;;
+     e2k:Linux:*:*)
+         echo e2k-unknown-linux-gnu
+         exit ;;
     frv:Linux:*:*)
         echo frv-unknown-linux-gnu
         exit ;;
```

Typical configure script run as follows:

```
./configure --build=e2k-unknown-linux-gnu
```

Dpkg has feature for auto-detect arch type of the host and target machine which is called dpkg-architecture. It uses dpkg internal config files, such as cputable. This one contains all Debian known CPU and consists of three columns: Debian name for the CPU, GNU name for the CPU and regular expression for matching CPU part of config.guess output. So we have to patch cputable too:

```
--- cputable-orig
+++ cputable-fix
@@ -34,3 +34,4 @@
     sh4      sh4          sh4
     sh4eb    sh4eb       sh4eb
     sparc    sparc       sparc(64)?
+e2k      e2k-unknown    e2k
```

Almost all packages use dpkg-architecture and get correct architecture identifier. If they don't, we fix it.

IX. CONCLUSION

This article is attempt to share experience in porting Debian on the architecture which is not supported by the community. General and specific for e2k architecture problems were described, as well as methods for their solutions. Authors hope that the article will be useful and interesting for developers, who support Debian on different architectures.

REFERENCES

- [1] Babayan B., "E2K Technology and Implementation", *Proceedings of the Euro-Par 2000 - Parallel Processing: 6th International*, Volume 1900/2000, pp. 18-21, January 2000.
- [2] Dieffendorf. K., "The Russians Are Coming. Supercomputer Maker Elbrus Seeks to Join x86/IA-64 Melee" *Microprocessor Report*, Vol. 13, №2, pp. 1-7, February 15, 1999.
- [3] Volkonskiy V., "Optimizing compilers for Elbrus-2000 (E2k) architecture", *4-th Workshop on EPIC Architectures and Compiler Technology*, 2005.
- [4] <http://www.edg.com/>
- [5] Hall J., "Distributed Compiling with distcc", *Linux Journal*, Issue 163, November 2007.

Generating environment model for Linux device drivers

Ilja Zakharov
ISPRAS

Moscow, Russian Federation
Email:
ilja.zakharov@ispras.ru

Vadim Mutilin
ISPRAS

Moscow, Russian Federation
Email: mutilin@ispras.ru

Eugene Novikov
ISPRAS

Moscow, Russian Federation
Email: novikov@ispras.ru

Alexey Khoroshilov
ISPRAS

Moscow, Russian Federation
Email:
khoroshilov@ispras.ru

Abstract— Linux device drivers can't be analyzed separately from the kernel core due to their large interdependency with each other. But source code of the whole Linux kernel is rather complex and huge to be analyzed by existing model checking tools. So a driver should be analyzed with environment model instead of the real kernel core. In the given paper requirements for driver environment model are discussed. The paper describes advantages and drawbacks of existing model generating approaches used in different systems of model checking device drivers. Besides, the paper presents a new method for generating model for Linux device drivers. Its features and shortcomings are demonstrated on the basis of application results.

Keywords—operating system; Linux; kernel; driver; model checking; environment model; Pi-processes

I. INTRODUCTION

Linux kernel is one of the most fast-paced software projects. Since 2005, over 7800 individual developers from almost 800 different companies have contributed to the kernel. Each kernel release contains about 10000 patches - work of over 1000 developers representing nearly 200 corporations [1]. Up to 70% of Linux kernel source code belongs to device drivers, and more than 85% errors, which lead to hangs and crashes of the whole operating system, are also in the drivers' sources [2] [3].

A. Linux device drivers

The Linux kernel could be divided into two parts - core and drivers (look at Fig. 1). Drivers manage devices and the kernel core is responsible for a process management, memory allocation, networking and et al.

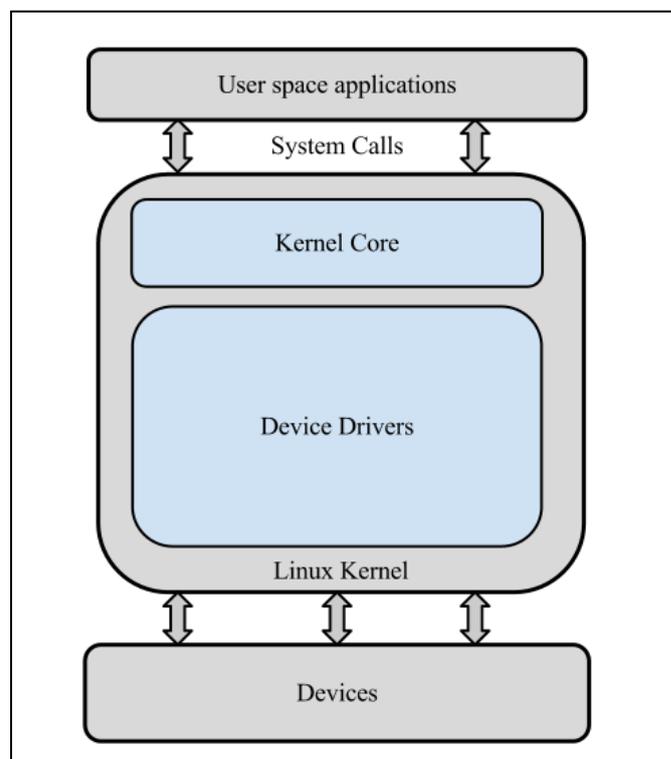


Fig. 1. Device drivers in the Linux kernel.

Most of drivers can be compiled as modules that can be loaded on demand. Drivers differ from common C programs. Drivers do not have a main function and a code execution order is primarily determined by the kernel core. Let us describe driver organization by considering a simplified example of a driver in Fig. 2:

- **Driver initialization function** (the function *init* below). A module of a driver is loaded on demand by the Linux kernel core when the operating system starts or when a necessity to interact with a corresponding device occurs. A module execution always begins with an invocation of a driver initialization function by the kernel core. In the Fig. 2 the initialization driver function is *usbpn_init*.

- **Driver exit function** (the function *exit* below). An interaction with the device is allowed until the module is unloaded. This happens after an invocation of a driver *exit* function by the kernel core. The function *usbpn_exit* is such function in Fig. 2.
- **Driver Handlers.** Various driver routines are usually implemented as callbacks to handle driver-related events, e.g. system calls, interrupts, et al. There are two handlers in the example in Fig. 2: *usbpn_probe* and *usbpn_disconnect*.
- **Driver structures** (we will call them just “structures” from now on). Most of handlers that work with common resources consolidated in groups. Each handler in such a group implements certain functionality defined by its role in this group. Usually pointers to handlers from one group stored in fields of a special variable with complex structure type. That is why we identify such groups as “driver structures”. In example in Fig. 2 *usbpn_driver* is the driver structure with *usb_driver* type. It has two fields “.probe” and “.disconnect” initialized with pointers to *usbpn_probe* and *usbpn_disconnect* handlers.
- **Registration and deregistration of handlers.** Before the kernel core can invoke handlers from the module, they should be registered. The typical way to register driver handlers is to call a special function. The function registers the driver structure with handlers and since the structure is registered, its handlers can be called. The driver structure registration takes place in the driver initialization (in *init* function body) or in an execution of a handler from another structure. An example of registration of *usb_driver* structure is illustrated in Fig. 2: *usb_register* is called in *usbpn_init* function body and it registers *usbpn_driver* structure variable. Also similar deregistration functions are implemented for the handler deregistration.

Even this simplified example illustrates the complexity of device drivers. A lot of driver methods are called by the kernel core such as handlers, *init* and *exit* functions and there are routines from the kernel core that are invoked by the driver such as register and unregister functions and other library functions. Besides, interaction of the kernel core and the driver depends on system calls from the user space and interrupts from devices. Their large interdependency with each other leads to availability of almost arbitrary scenarios of handler calling. But in all of such scenarios rules of correctness are taken into account such as restrictions on order, parameters and context of handler invocations.

B. Model checking Linux device drivers

Nowadays it is not easy to maintain the safety of all device drivers manually due to complexity of drivers, high pace of the Linux kernel development and huge size of source code. That is why an automated driver checking is required. There are various techniques for achievement this goal and a model checking approach is one of them.

```

static int usbpn_probe(struct usb_interface *intf, const
struct usb_device_id *id){
...
}
static void usbpn_disconnect(struct usb_interface *intf){
...
}
static struct usb_driver usbpn_driver = {
.name = "cdc_phonet",
.probe = usbpn_probe,
.disconnect = usbpn_disconnect,
};
static int __init usbpn_init(void){
return usb_register(&usbpn_driver);
}
static void __exit usbpn_exit(void){
usb_deregister(&usbpn_driver);
}

```

Fig. 2. A simplified example of a driver drivers/net/usb/cdc-phonet.c¹ (compiled as cdc-phonet.ko module).

As illustrated before, a driver execution depends on the kernel core. But analysis of a driver together with the kernel core is rather difficult nowadays for tools due to complexity of kernel core source code and its huge size. That is why a driver environment model is required for analyzing device drivers. The model can be implemented as C program that emulates interaction of the driver with the kernel core. In general the model should emulate the interaction with hardware too, but this aspect is not considered in this paper. The driver environment model should provide:

- Invocation of the driver initialization and exit functions.
- All available in the real interaction of the kernel core and the driver scenarios of invocations of handlers taking into account:
 - Limitations on parameters of handler calls.
 - A context of handler invocation such are interrupts allowed or not.
 - Limitations on order and number of invocations of handlers for:
 - Handlers from a driver structure.
 - Handlers from different driver structures.
- Models for kernel core library functions.

An incorrect model often causes a false positive verdict from a verification tool (*verifier* below) or a real bug skipping

¹ <http://lxr.free-electrons.com/source/drivers/net/usb/cdc-phonet.c?v=3.0>

[4]. An example of environment model for the driver considered above is shown in Fig. 3:

```

void entry_point(void){
    // Try to initialize the driver.
    if(usbpn_init())
        goto final;
    // The variable shows usb_driver device is probed
    // or not.
    int busy = 0;
    // For call sequence of handlers of any length.
    while(1){
        // Nondeterministic choosing
        switch(nondet_int()){
            case 1:
                // The device wasn't probed.
                if(busy == 0){
                    res = usbpn_probe(..);
                    if(res == 0){
                        busy = 1;
                    }
                }
                break;
            case 2:
                // The device was already
                // probed.
                if(busy == 1){
                    usbpn_disconnect(..);
                    busy = 0;
                }
                break;
            case 3:
                // Try to unload the module
                // if the device wasn't probed.
                if(busy == 0){
                    goto exit;
                }
                break;
            default: break;
        }
    }
    // Unload driver.
    exit: usbpn_exit();
    final:
}

```

Fig. 3. Environment model for the driver from Fig. 2.

A verifier starts the driver analysis from the function *entry_point*. First the driver should be initialized by the function *ubpn_init*. If it returns success result “0”, then *usbpn_probe* and *usbpn_disconnect* are invoked. The variable *busy* is needed for calling handlers in the proper order. The handler *usbpn_probe* should be called the first and if it returns success result, then *usbpn_disconnect* should be called. Operator *while* is needed for call sequences of handlers of variable length. Operator *switch* with the function *nondet_int* returning random *int* provides non-deterministic handler

calling for various scenarios of handler invocations covering in the. At the end of a driver work *usbpn_exit* should be called, but it can take place only if the device wasn't probed, either when *usbpn_probe* wasn't called or when *usbpn_probe* returned an error value or when *usbpn_probe* and *usbpn_disconnect* were already called one or more times. After the *exit* invocation a verifier finishes analysis.

II. RELATED WORK

There are several verification systems for device drivers, but only Microsoft SDV [5] is in industrial use. For modeling driver environment various approaches are used.

- **Microsoft SDV**. SDV provides a comprehensive toolset for analysis of source code of device drivers of Microsoft Windows operating system. These tools are used in the process of device driver certification, and have been included in Microsoft Windows Driver Developer Kit since 2006. SDV's driver environment model is based on manually written annotations of handlers. SDV provides a kernel core model that contains simplified stubs of some kernel core routines. Microsoft SDV is specifically tailored for analysis of device drivers of Microsoft Windows. Unfortunately, it is proprietary software, which prohibits its application to other domains outside Microsoft.
- **Avinux** [6]. This project was developed in University of Tubingen, Germany. Its environment model is based on handwritten annotations of each handler. Authors paid attention to the problem of proper initialization of various resources and uninitialized pointers in the environment model [7].
- **DDVerify** [8]. The project was developed in Oxford and Carnegie Mellon universities. Authors implemented a partial kernel core model for a special kernel version for verifying drivers of several types. But the model is handwritten and maintaining it manually is complicated while the kernel is under continuous development.
- **LDV** [9]. LDV framework for driver verification is developed in Institute of System Programming of Russian Academy of Sciences. This project took a high pace of the Linux kernel development. An environment model generation process is fully automated and does not need manual annotations in code. It is based on an analysis of the driver source code and on a configuration. The configuration consists of handwritten specifications for several driver structures and a heuristic template for other cases. Generated model provides nondeterministic handler call sequences, interrupt handlers invoking. A model can be generated for a driver module from any subsystem and in most cases it correctly describes interaction of a driver with the kernel core.

The lack of such sufficient handicaps as a demand for handwritten annotations or the difficulty of model maintaining allows to efficiently using LDV for verifying all kernel drivers

from a lot of kernel releases. However, a driver environment generator has considerable limitations:

- Only linear handler call sequences are available, where each handler can be called only once.
- Driver structures registration and deregistration are not taken into account in model generation process.
- Source code analysis is based on regular expressions. This approach leads to syntactic mistakes in a model due to changes in the kernel and complexity of the kernel source code.
- Not all needed restrictions on handler calls can be described in the configuration.
- For a driver module that consists of several files the tool generates separate models for each “.c” file but not the one for the whole module.

Such shortcomings lead to incorrect verdict from a verifier or real bugs missing. And in most cases the tool doesn't provide any capabilities for overcoming model imperfectness. This paper suggested a new approach for generating driver environment model.

III. SUGGESTED APPROACH

The main goal of this research was to develop a new tool for automatically generating environment model for kernel driver modules which contain one or several files. An approach suggests environment model that should take into account:

- All available in real driver handler call scenarios from a one driver structure.
- Limitations on order of calling handlers from several driver structures.
- Association of handlers invocation with registrations and deregistration of driver structures.
- Restrictions on handler call parameters.

A new generator should provide full automated generating of environment model for a kernel driver module and facilities for describing restrictions on handler calls in the model.

Moreover the tool should provide additional capabilities for driver environment model debugging, altering generated code and understanding scenarios available in generated C code.

IV. ARCHITECTURE OF THE NEW DRIVER ENVIRONMENT MODEL GENERATOR

The design of the new driver environment model generator (DEG) is illustrated in Fig. 4. An input file for DEG is a LDV command stream. This file contains information on build options for compiler, paths to driver and kernel source code, et al. LDV components connects with each other through this file and DEG transforms it during its work. The environment model generation process consists of 3 steps: driver source code analysis, generating of the model in the intermediate

representation and printing of corresponding C code. We shall consider these steps below in details.

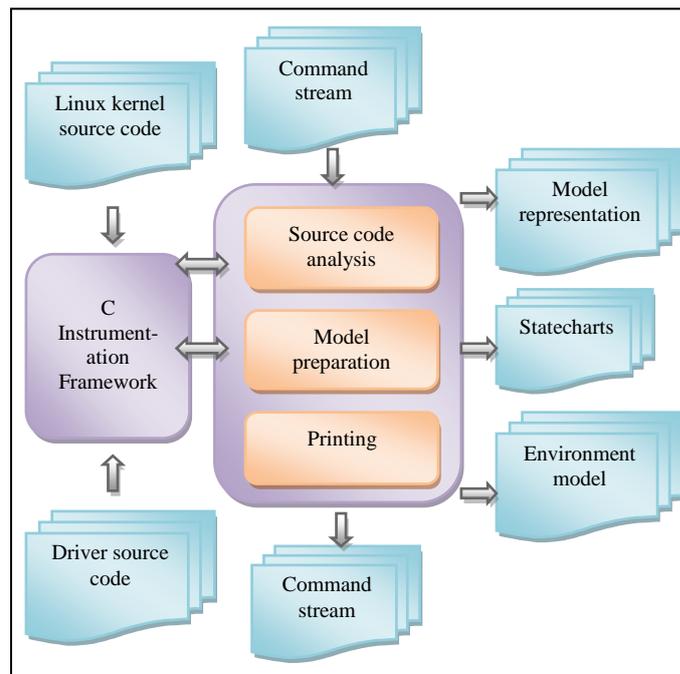


Fig. 4. Design of the driver environment generator.

A. Driver source code analysis

Linux kernel source code is sophisticated and often changing. That is why using analysis based on regular expressions leads to various bugs in generated code of the model or lack of information on source code for model generation. For solving this problem DEG uses C instrumentation framework (CIF below) for source code querying [10]. DEG requests information from this tool about driver source code like initialization and exit procedures, driver structures, library function invocations, et al. Querying process can be divided into two steps:

- 1) Querying for handlers and driver structures used in the driver.
- 2) Querying for functions used for registration of these driver structures and other queries based on information extracted at a first step.

After source code analysis it is needed to get additional information on handler call order, handler return values and handler arguments before environment model can be generated. Such the information is stored in a configuration.

B. Internal driver environment model representation construction

Paper [11] designed a formal driver environment model based on Robin Milner's Pi-processes [12]. The model is considered as a parallel composition of Pi-processes. A group of handlers from one driver structure corresponds to a Pi-process. Interactions between such processes are implemented by signals exchanging. Driver structure registration and

deregistration are modeled via these signals too. Also this work proposed a method of translating such driver environment model into a multi-threaded C program. And it showed that the translated sequential program reproduces the same traces as available in the initial model via Pi-processes. This result is important because nowadays model checking verifiers don't support multi-threaded C programs analysis but drivers can be executed in several threads.

A DEG configuration is developed for specifying Pi-processes of environment model. The configuration consists of two parts: manually written specifications for several driver structures and patterns for automatically generating such specifications for other driver structures. This design of configuration allows generating Pi-process description for difficult cases using manually written specifications and for other cases using patterns.

New DEG constructs a model representation on the basis of the configuration and data extracted from the source code. The following algorithm for constructing the representation is used:

- 1) First of all presence of manually written descriptions in configuration is checked for each driver structure that was found in the driver.
- 2) If such specification for a driver structure is found, it will be adopted for this driver structure taking into account its handlers and registration methods that were founded in the driver source code. This adopted specification is used for modeling the driver structure in the model representation.
- 3) If a description is not found, then a suitable pattern will be chosen from the second part of the configuration. This pattern will be adopted using heuristics and taking into account the driver structure.

As a result of this stage DEG provides a driver environment model representation with Pi-processes descriptions for each driver structure found in the driver source code and signals that are used for interaction between processes. Representation format almost coincides with the format of the configuration.

For debugging purposes DEG can generate statecharts with available handler invocation scenarios in the generated code. Each statechart illustrates the call handler order for the corresponding Pi-process. Simplified examples of such charts showed in Fig. 5 and Fig. 6. These figures illustrate two graphs for order of calling *init* and *exit* functions and for order of calling handlers from the driver structure with the *usb_driver* type. In the Fig. 5 there are 3 states: "state 0" in which driver wasn't been initialized yet, "state 1" in which driver normally operates and "state 2" in which it is already unloaded. In "state 1" other handlers can be called like it is illustrated in the charts in the Fig. 6: after *init* execution, *usb_driver* structure is become registered, after this event handler *probe* can be called. If the device was probed successfully handler *disconnect* can be called. After *disconnect* invocation *exit* can be called that unregisters *usb_driver* structure.

C. Printing C code of driver environment model

On the last stage DEG translates the driver environment model representation based on Pi-processes into a C code.

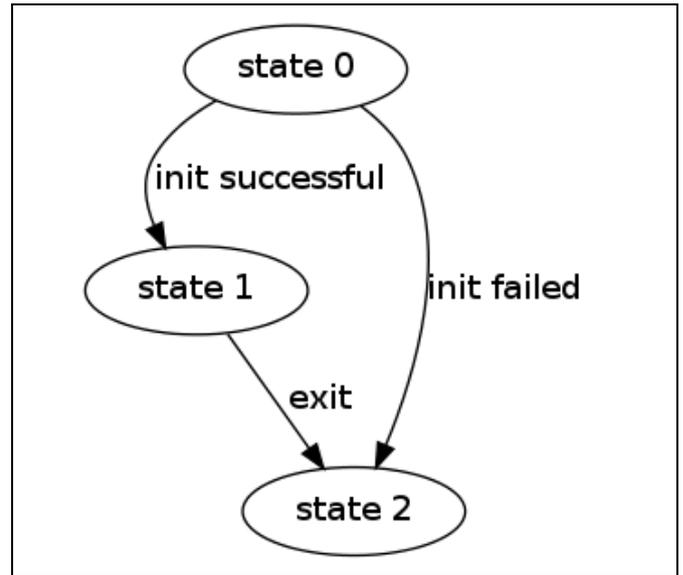


Fig. 5. Simplified example of the statechart for *init* and *exit* functions.

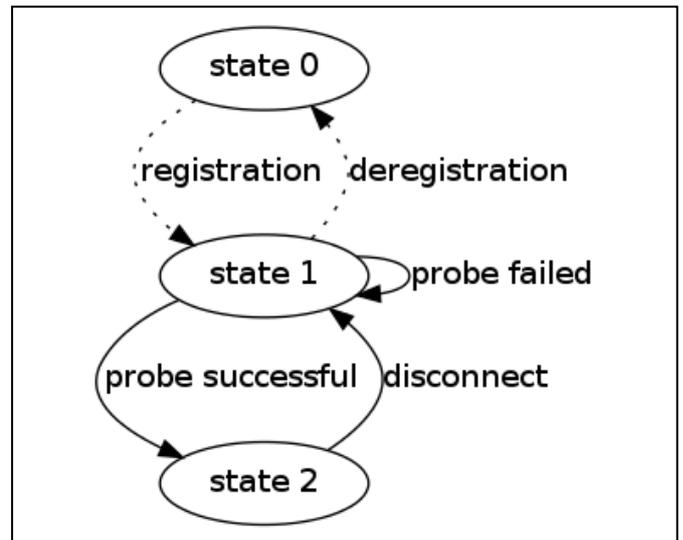


Fig. 6. Simplified example of the statechart for handlers from *usb_driver* driver structure.

Then DEG represents this C code in form of aspect files, which are used by another LDV component Rule Instrumentor. After applying these aspect files by that component, code of the generated model is added to the driver source code and some driver routines are also changed. Added code includes various auxiliary routines, variables and *entry_point* function in which handlers are invoked and from which a verifier starts its analysis.

V. RESULTS

New DEG is still under development, but some results have been obtained already. For a comparison of the new tool with the old one 2672 drivers from the Linux kernel 3.8-rc1 were analyzed. As a model checking verifier BLAST tool was used [13].

Table I illustrates transitions of verification verdicts after switching to the new driver environment model generator. Columns show results of checking of modules for a corresponding error type connected with: blk_requests executing (1); classes, chrdev_regions and usb_gadgets allocating (2); pairing of *module_get* and *module_put* routines (3); using locks (4). The first table line contains the numbers of modules without any exposed errors for both old and new DEG. One of the main goals of development of the new tool was to decrease the number of false positives from a verifier due to incorrect environment model. Progress in this direction is illustrated in the second line. The number of transitions isn't as much as expected due to incorrect work of other LDV components and BLAST tool or time and memory limits (because more resources are needed for proving safety of a driver than for finding an error). Next three lines demonstrate cases with true and false positives from the verifier. The first of these lines illustrates the number of modules whose environment model becomes better. In the next line there are cases with still incorrect environment model. And the last of these 3 lines stores true positives or false positives with the incorrect verdict occurred not due to environment model (for example due to an imperfect pointer analysis by BLAST). Next 2 lines contain the number of absence of the verdict with a new model. In 20% of these cases a reason is limit of memory or time because in some cases new DEG generates a sophisticated and large model. In other cases reasons are various bugs in LDV or in the verifier. Next two lines contain number of cases when an old model had syntax errors in the contrast to the new one. The last line shows cases with LDV or verifier fails despite both new and old environment model because these modules are too huge or just due to bugs in verification system or in the verifier.

TABLE I. VERIFICATION VERDICTS AFTER SWITCHING TO THE NEW DRIVER ENVIRONMENT MODEL GENERATOR.

Transitions	1	2	3	4	
Safe → Safe	2469	2441	2414	2444	
Unsafe → Safe	0	2	5	7	
Unsafe → Unsafe	<i>Model becomes better</i>	6	3	6	4
	<i>Model is still incorrect</i>	0	1	6	5
	<i>Unsafe is not due to model</i>	0	15	18	5
Safe → Unknown	43	44	46	45	
Unsafe → Unknown	0	1	16	4	
Unknown → Safe	12	13	10	16	
Unknown → Unsafe	0	1	4	1	
Unknown → Unknown	142	151	149	141	

Proper environment model is one of necessary conditions for obtaining true verdicts. Despite a minor number of transitions from false positives, an experience of using the new generator showed that such incorrectness of an environment model often hides various problems in other LDV components or in verifier. Switching to the new generator explored such problems and allowed to increase quality of driver verification in general.

VI. FURTHER DEVELOPMENT DIRECTIONS

The suggested approach increased quality of generating of driver environment models, but there are the following shortcomings in the current tool that should be solved in future:

- **Configuration extension.** For several types of drivers specifications for driver structures should be written manually in the configuration. The number of driver structures is estimated as two hundreds in the whole kernel. There are 15 described already in the configuration and about 15 are needed to be specified.
- **Interrupts, timers, tasklets modeling.** New DEG doesn't invoke interrupt handlers, timer routines or tasklet callbacks yet. For increasing coverage of code analysis they should be invoked in the new model.
- **Generating model for several modules.** Sometimes an analysis of only one module leads to sophisticated or incorrect environment model, because drivers can contain several modules or common routines from several drivers are picked out to a library module. Thus environment model should be generated for groups of interacting modules rather than for separate modules of these groups.

VII. CONCLUSION

The paper describes the new approach for automatically generating driver environment models for model checking Linux kernel drivers. Also it demonstrates the new version of the component of LDV framework called Driver Environment Generator implementing this approach. The new DEG provides:

- Fully automated environment model generating for drivers that can be compiled as Linux kernel modules. Generating process is based on source code analysis performed by C Instrumentation Framework [10].
- The new configuration for generating process management. This configuration consists of specifications for driver structures and patterns for invoking handlers from other driver structures having an unknown type. The configuration is based on Pi-processes and allows setting various restrictions for handler invocation including restrictions on order and parameters of calling handlers from one or several driver structures.
- Facilities for simplifying work with generated environment models by its representation in

configuration format or statecharts that illustrate order of handler calls.

- Driver environment model as a set of aspect files for applying to the driver source code by LDV component Rule Instrumentor.

Initial experience of the new tool application demonstrated that the new approach allows increasing quality of generated environment models and decreasing the number of false positives from verifiers. Also usability of DEG tool was improved.

The new DEG will replace soon the old one and will be available as component of LDV framework. Information on LDV framework is available on the site of the project <http://linuxtesting.org/project/ldv>.

References

- [1] J. Corbet, G. Kroah-Hartman, A. McPherson., "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] A. Chou, J. Yang, B. Chelf, S. Hallem, and DR Engler, "An Empirical Study of Operating System Errors," Proceedings of the 18th ACM Symp. Operating System Principles, 2001.
- [3] M. Swift, B. Bershad, H. Levy, "Improving the reliability of commodity operating systems," Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.
- [4] D. Engler, M. Musuvathi, "Static analysis versus model checking for bug finding", Proceedings of the 16th international conference CONCUR 2005, San Francisco, CA, USA, 2005.
- [5] T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg, "The Static Driver Verifier Research Platform," Formal Methods in Computer Aided Design, 2010.
- [6] H. Post, W. Kuchlin, "Integrated Static Analysis for Linux Device Driver Verification," Proceedings of the 6th international conference on Integrated formal methods, Germany, 2007.
- [7] H. Post, W. Kuchlin, "Automatic data environment construction for static device drivers analysis," Proceedings of the conference on Specification and verification of component-based systems, USA, 2006.
- [8] T. Witkowski, N. Blanc, D. Kroening , G. Weissenbacher, "Model Checking Concurrent Linux Device Drivers," Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM, USA, 2007.
- [9] M. Mandrykin, V. Mutilin, E. Novikov, A. Khoroshilov, P. Shved, "Using linux device drivers for static verification tools benchmarking," Programming and Computer Software September 2012, Volume 38, Issue 5, pp 245-256.
- [10] A. Khoroshilov, E. Novikov, "Using Aspect-Oriented Programming for Querying Source Code," Proceedings of the Institute for System Programming of RAS, volume 23, 2012.
- [11] V. Mutilin, "Verification of Linux Operating System Device Drivers with Predicate Abstractions," Phd's Thesis, Institute for System Programming of RAS, Moscow, Russia, 2012.
- [12] R. Milner, "A Calculus of Communicating Systems," Springer-Verlag (LNCS 92), ISBN 3-540-10235-3, 1980.
- [13] D. Beyer, T. Henzinger, R. Jhala, R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering," International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007.

On the Implementation of Data-Breakpoints Based Race Detection for Linux Kernel Modules

Nikita Komarov
ISPRAS
Moscow, Russia
nkomarov@ispras.ru

Abstract—An important class of problems in software are race conditions. Errors of this class are becoming more common and more dangerous with the development of multi-processor and multi-core systems, especially in such a fundamentally parallel environment as an operating system kernel. The paper overviews some of existing approaches to detect race conditions including DataCollider system based on concurrent memory access tracking. RaceHound, a race condition detection system for Linux drivers based on similar principles as DataCollider is presented.

Keywords—driver verification; race condition; linux kernel; dynamic verification; operating system

I. INTRODUCTION

The Linux Kernel is one of the most popular and fast-developed projects in the world. Linux Kernel development started in 1991 by Linus Torvalds. The development process of Linux kernel is distributed, about 1,000 people worldwide are involved in the preparation of each new kernel release. The new release comes out every 2-3 months. Changes are submitted by the developers in the form of little pieces of code called patches. Each kernel release consists of about 9-13 thousands of patches, which corresponds to an average of about 7.3 patches per hour. The total source code size of one of the latest versions of the Linux kernel - version 3.2 - is about 15 million lines. These data are given in the latest Linux Foundation report on Linux Kernel development [6].

Linux Kernel development process is described in [7]. There are some other branches of kernel development based on the original Linux Kernel. Some of the Linux distributions developers support their own versions of the kernel - for example, Red Hat [11], openSUSE [12] and Debian [13]. These kernels are different from the original version in that they support some additional functionality and/or contain bug fixes. There are some kernel versions with the significant changes to the basic systems of the kernel, for example, a real-time Linux Kernel [14] or the Android kernel [15]. Over time some changes from different branches of development needed by a broad range of people can get to the original kernel.

As with any programs, there are various errors in Linux Kernel that lead to the incorrect functioning of the OS, freezing etc. The greatest part of the kernel (about 70%) are various device drivers. The results of studies that have been carried out in [16] and [17] in the early 2000s for kernels 1.0 to 2.4.1, showed that drivers contain up to 85% of all errors in the Linux Kernel. A similar study for the Microsoft Windows XP kernel in 2006 also showed that the highest number of errors in the operating system kernel belongs to the device drivers [18]. More recent studies done in 2011 for the Linux Kernel versions from 2.6.0 to 2.6.33 showed that although the number of errors in the drivers became less than in the kernel components responsible for the support of the various architectures and file systems, their share is still high [19].

The task of ensuring the reliability of drivers is important as drivers in Linux work with the same privileges level as the rest of the kernel. Because of this a vulnerability in the drivers can lead to the possibility of execution of arbitrary code with kernel privileges and access to the kernel structures.

II. RACE CONDITIONS

One of the important types of errors in the software is race conditions [20]. A race condition occurs when a program is working wrong due to an unexpected sequence of events that leads to the simultaneous access to the same resource by multiple processes.

As an example of a race condition, consider a simple expression in some programming language: $b = b + 1$. Imagine that this expression is executed simultaneously by two processes, the variable b is common to them, and its initial value is 5. Here is a possible example of the order of execution of the program:

- Process 1 loads b value in a register.
- Process 2 loads b value in a register.
- Process 1 increases its register value by 1 with a result of 6.
- Process 2 increases its register value by 1 with a result of 6.
- Process 1 stores its register value (6) in the variable b .

- Process 2 stores its register value (6) in the variable b.

The initial value of b was 5, each of the processes added 1, but the final result was 6 instead of the expected 7. Processes are not executed atomically, another process may intervene and perform operations on some shared resource between almost any two instructions. Similarly, the classic example is the simultaneous withdrawal of money from a bank account from two different places: if the check for the required amount in the account by the second process would occur between similar check and amount decrease of the first process, the account balance may become wrong that will cause a loss and significant reputation damage.

With the development of multicore and multiprocessor systems race condition related errors including the Linux Kernel, becomes even more important than before. For example in the study [1] it was concluded that the race conditions are the most frequent type of error in the Linux Kernel and make up about 17% of typical errors in the Linux Kernel (on the second and third place are specific objects leaks and null pointer dereference – 9% both). The study was conducted by analyzing the comments to the changes in the Linux Kernel. From the above it can be concluded that race conditions are an important and common class of errors, including those in the Linux Kernel, and the task to find them is relevant.

III. EXISTING METHODS FOR DETECTING RACE CONDITIONS

There are various ways to detect race conditions in programs. Most of the dynamic methods are based on two principles: Lockset and Happens-before [4] [5]. Lockset based tools check if there is synchronization between threads when accessing shared variables. This makes it possible to find a large number of potential errors, but the number of false positives is high too.

Happens-before based instruments find accesses from different threads to a specified area of memory that have no specified order, meaning they can be in a different order. These instruments depend on how the access is done in a real system operation, so they identify a smaller subset of errors but have greater accuracy than the Lockset method. An alternative to this method is a direct test for simultaneous memory access by placing breakpoints - this method is implemented in the DataCollider system (see sect. II.C). In most real systems, a combination of two methods is used. Let's consider some examples of such systems.

A. Helgrind

Helgrind is a tool for analyzing user mode programs for race conditions, based on the Valgrind framework [10]. This system can detect three types of errors:

- Improper pthreads API use;
- Possible deadlocks that occur due to incorrect order of synchronization mechanisms;
- Race conditions.

Helgrind detects race conditions by monitoring all accesses to the memory of the process and all use of synchronization primitives. Then the system builds a graph, based on which it makes a conclusion that there is a «happens-before»

relationship between accesses. If the access to a certain area of memory happens in two different threads, the system checks whether there can be found the «happens-before» relationship between them, that is, whether one of accesses happen before the other. The system makes conclusions of the presence or absence of such a connection based on the presence or absence of various synchronization primitives. If the memory access occurs in at least two different threads and the system cannot find a relationship «happens-before» between them, it concludes that there is a data race between them.

B. ThreadSanitizer

ThreadSanitizer is another engine that finds race conditions in user space programs. The algorithm of this system is similar to the Helgrind algorithm and is described in [23]. The system instruments the program code adding calls to its functions before each memory access and every time the program uses some synchronization tools. The system then tries to figure out which of the memory accesses occur with inadequate synchronization and may conflict with other memory accesses. ThreadSanitizer also has an offline mode in which it can be used to analyze traces created by some other tools such as Kernel Strider for Linux Kernel [24].

C. DataCollider

The system is designed for dynamic race conditions detection in Microsoft Windows kernel. It was developed in Microsoft Research and is described in [3]. The system uses the principle which is slightly different from other described systems and is as follows:

- The system periodically sets up software breakpoints in random places of studied code.
- When the software breakpoint is triggered, the system decodes the triggered instruction getting the memory address and sets a hardware breakpoint on access to this address. Then it stops the process execution for a short time to increase the chance of another access to this address.
- After the delay the system removes the hardware breakpoint.
- If the hardware breakpoint is triggered, data race is reported. Also data race is reported if the value at the address has changed – to take the possible use of direct memory access (DMA) into account.

DataCollider system is used in Microsoft, it helped to find about 25 errors in the Windows 7 kernel. In [3] low overhead advantage of the system is noted: it can find some errors in the kernel even with the settings causing overhead of less than 5%.

IV. LINUX DRIVER VERIFICATION

The main features of driver design and verification are the direct work with the hardware, a common address space, limited set of user space interfaces and multithreading. This makes it difficult to debug the drivers and to determine the causes of errors. Let's consider some software products that are used to verify the Linux drivers.

A. *Kmemleak, kmemcheck*

These systems are the most known and widely used. They are included in Linux Kernel. *Kmemleak* [8] is a system for finding memory leaks. Its principle of operation is similar to those in some of the garbage collectors in high level languages. For every memory allocation, the information about the selected memory area (address, size etc.) is stored, and when this area is deallocated the corresponding entry is removed. The system can be interacted with via a character device in debugfs. With every access to this device the following steps are conducted:

- The "white" list of the allocated and not freed memory areas is created.
- Certain areas of memory are scanned for pointers to the memory of the "white" list. If the system finds such a pointer, the memory is transferred from the "white" list to the "gray" list. The memory in the "gray" list is considered accessible and not leaked.
- Each block of the "gray" list is also scanned for pointers to the memory of the "white" list.
- After this scanning all memory left in the "white list" is considered to memory leaks.

Kmemcheck [9] is a simple system that keeps track of uninitialized memory areas. Its principle of operation is as follows:

- The system intercepts all the memory allocation. Instead of each area an area 2 times as big is allocated, these additional ("shadow") pages are initialized with zeroes and are hidden.
- The allocated memory area is returned to the caller with cleaned "present" flag. As a result, any reference to this memory will result in page fault.
- When such a memory access happens, *kmemcheck* determines the address and size of the corresponding memory access. If the access is for writing, the system populates the corresponding bytes of "shadow" page with 0xFF, then successfully completes the operation.
- If the access is for reading, the system checks the appropriate bytes of "shadow" pages. If at least one of them is 0, uninitialized memory access is reported.

B. *KEDR*

KEDR (short for *KErnel-mode Drivers in Runtime*) is a system for the dynamic analysis of Linux Kernel modules [21]. This system can replace some kernel function calls with its wrappers which can produce some additional actions such as saving the information of function calls or just returning errors. Users can create their own systems based on this system, and solutions for some specialized tasks are included, such as:

- Memory leaks detection. To solve this problem, the system keeps track of calls to different functions that allocate and free the memory. After unloading the tested module the system creates a report containing all memory locations that have been allocated, but have not been freed, along with the call stack for each

of the memory allocation functions calls. The report also includes any attempts to free the memory that has not been allocated. The scenario is different from the *Kmemleak* system in that the memory leak detection happens after the unloading of the target module, which simplifies the algorithm.

- Fault simulation. To solve this problem given functions are replaced by a wrapper which returns errors on defined scenario.
- Call tracking. Information about calls to given functions (including arguments, return values etc.) is stored in a file for later analysis.

This system has been used successfully and helped to find about 12 errors in various Linux drivers [22].

C. *Static methods*

Static program verification is the analysis of program code without actually executing it, as opposed to dynamic analysis. It does not require setting up the test environment, and provides the ability to analyze all the possible execution paths of the program, even those which need the coincidence of several rare conditions. When applied to the OS Kernel, static verification is particularly useful because in many cases creating a test environment and analyzing some of the execution paths can be a non-trivial task. However, static analysis has many limitations. The main part of the paper is devoted to the dynamic verification system, so we will not examine the static verification methods in detail. A more detailed review of these methods is given in [2].

V. RACEHOUND

As part of the Google Summer of Code 2012 [25], the author developed a lightweight race detection system for Linux Kernel. The algorithm used by this system is similar to the algorithm used by the *DataCollider* system (see sect. II.C). The system is designed not only to find race conditions, but also to confirm the data obtained with other systems that can produce false alarms, for example, *ThreadSanitizer* (see section II.B). At present the system supports the x86 and x86-64.

The originally planned principle is as follows:

- The system randomly plants software breakpoints (there is a Linux Kernel API called *Kprobes* [26] for that) in various places of the investigated kernel module, periodically changing them.
- When the software breakpoint is triggered, the system decodes the instruction on which the breakpoint was planted, using the decoder of the Linux Kernel modified in *KEDR* project. Then it determines the memory address which the instruction tries to access and sets the hardware breakpoint on this address (there is also an API in Linux Kernel for this [27]), and then stops the process for a short time to increase the chance of access from another process to this address.
- After the delay the system removes the hardware breakpoint.

- If the hardware breakpoint was triggered in the elapsed time, the race condition is reported. The race condition is also reported if the value at the address has changed – to cover the case of direct memory access.

Software breakpoints in x86 architecture work as follows. The first byte of instruction at the specified address is replaced by the 0xCC byte – interrupt INT3 – preserving the original byte in some place. When the CPU executes the instruction, an interrupt is triggered and control is passed to the interrupt handler in Linux Kernel, which searches the list of software breakpoints for the appropriate address. When the address is found, it transfers control to the appropriate handler. After the handler finishes the original instruction is executed.

Hardware breakpoints are implemented as four debug registers on Intel x86 processors. This system is described in the Intel Developer Manuals [28]. An addresses can be written in these registers, and there will be an interrupt on an access to these addresses. The interrupt is then processed by the Linux Kernel which transfers control to the appropriate hardware breakpoint handler.

A. Implementation features

There were some problems in the implementation of system. Software breakpoint handlers are executed in an atomic context, and therefore it was impossible to properly set the hardware breakpoint for all available CPUs. This problem was solved by installation and removal of hardware breakpoint not from the software breakpoint handler, but from the function in the task queue. Unfortunately, this decision led to a time gap between the beginning of the delay and the hardware breakpoint setup, and therefore may reduce the probability of concurrent memory access to occur within the delay (for a very small time delay - down to 0) and to lower detection accuracy. This effect, however, requires a special study.

Another problem was the execution of the original instruction in the software breakpoint handler. This execution takes place inside an interrupt handler, but the original instruction refers to the address on which the hardware breakpoint has just been set. In some cases, removal of hardware breakpoint does not yet happen at the time of the original instruction execution, and the hardware breakpoint was triggered. However, the software breakpoint handler works in an atomic context and the interrupt is forbidden. This caused some faults and unusual behavior. This problem was solved by dropping Kprobes API and implementing similar functionality manually. Instead of executing the original instruction separate from the module code this instruction was restored and the control was transferred to the investigated module. To reset the breakpoints after that the timer has been set, which reset the breakpoints at frequent intervals replacing their first bytes with 0xCC. This decision, however, also has a drawback: there is a period in which a software breakpoint is not set at the needed place. This can also reduce the accuracy of error detection.

The system consists of a kernel module which has an interface based on the debugfs and some auxiliary scripts. The interface is a character device in debugfs which allows a user to set the possible breakpoints range, from which N is randomly chosen, in the format `<function name>+<offset>`. If the both parameters or just the offset are equal to *, the

complete module or the complete function is added, respectively.

An important limitation of the system is the inability to work on single-core systems. This problem is caused by the software breakpoint handler execution: it is performed in an atomic context, so putting the process to sleep is impossible. Therefore, instead of the function `msleep()` the `mdelay()` function is used, which waits a specified period of time, leaving the thread in running state. For this time no other tasks can run on the same processor. Therefore the process which could cause a race condition should run on another core to be able to execute at a delay time.

The system requires a Linux Kernel 2.6.33 or later (this version introduced the Hardware Breakpoints API). The build system is based on CMake. At present the system is in the state of working prototype. It requires testing on some real drivers to identify potential errors and defects, adjust the parameters of the system (number of breakpoints, time intervals, etc.) and to evaluate the effectiveness of the system in the real world. For testing it is necessary to pay attention to the choice of test cases for drivers – they should include some parallel and concurrent testing, because the system just increases the probability of errors, being useless if the concurrent access is impossible.

Another direction of the system development may be the development of interfaces and integration with other race condition detection systems. For example, the system can be useful when working together with static methods which provide a significant number of false positives to confirm these data with them. However, dynamic methods which can produce some false positives can also benefit from such integration.

VI. CONCLUSION

Race conditions are an important problem. This paper reviews some methods for detecting race conditions, including those in the operating system kernel, and some features of Linux drivers verification. The race condition detection system created by author is described.

Most of the race condition detection systems are based on one of the two methods: LockSet and Happens-before, or some kind of their combination. From a theoretical point of view, the direction of future work could be some more detailed review of existing methods for detecting race conditions in order to integrate the developed system with some of them.

Directions of further practical work should be testing the developed system on some real drivers and its integration with other systems, including those based on static methods. Testing on real drivers will help to identify errors or omissions, find some valid settings of various parameters of the system (number of breakpoints, time intervals etc.) and to evaluate the effectiveness of the system in real conditions.

- [1] V. Mutilin, E. Novikov, A. Khoroshilov
Analysis of typical errors in Linux operating system drivers.
Proceedings of Institute for System Programming of RAS, vol.22, 2012
- [2] M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. V. Khoroshilov, and P. E. Shved
Using Linux Device Drivers for Static Verification Tools Benchmarking
Programming and Computer Software, 2012, Vol. 38, No. 5

- [3] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, Kirk Olynyk
Effective Data-Race Detection for the Kernel
9th USENIX Symposium on Operating Systems Design and Implementation, 2010
http://static.usenix.org/event/osdi10/tech/full_papers/Erickson.pdf
- [4] Cormac Flanagan, Stephen N. Freund
FastTrack: Efficient and Precise Dynamic Race Detection
<http://slang.soe.ucsc.edu/cormac/papers/pldi09.pdf>
- [5] Nels E. Beckman
A Survey of Methods for Preventing Race Conditions
http://www.cs.cmu.edu/~nbeckman/papers/race_detection_survey.pdf
- [6] Jonathan Corbet, Greg Kroah-Hartman, Amanda McPherson
Linux Kernel Development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It (2012)
<http://go.linuxfoundation.org/who-writes-linux-2012>
- [7] Jonathan Corbet
How to Participate in the Linux Community. A Guide To The Kernel Development Process (2008)
<http://www.linuxfoundation.org/content/how-participate-linux-community>
- [8] Jonathan Corbet
Detecting kernel memory leaks
<http://lwn.net/Articles/187979/>
- [9] Jonathan Corbet
kmemcheck
<http://lwn.net/Articles/260068/>
- [10] Valgrind Manual. 7. Helgrind: a thread error detector.
<http://valgrind.org/docs/manual/hg-manual.html>
- [11] S. M. Kerner
The Red Hat Enterprise Linux 6 Kernel: What Is It? (2010)
<http://www.serverwatch.com/news/article.php/3880131/The-Red-Hat-Enterprise-Linux-6-Kernel-What-Is-It.htm>
- [12] OpenSUSE Kernel
<http://en.opensuse.org/Kernel>
- [13] Debian Kernel
<http://wiki.debian.org/DebianKernel>
- [14] OSADL Project: Realtime Linux
<https://www.osadl.org/Realtime-Linux.projects-realtime-linux.0.html>
- [15] Jonathan Corbet
Bringing Android closer to the mainline
<https://lwn.net/Articles/472984/>
- [16] A. Chou, J. Yang, B. Chelf, S. Hallem, DR Engler
An Empirical Study of Operating System Errors. Proc. 18th ACM Symp. Operating System Principles, 2001
- [17] M. Swift, B. Bershad, H. Levy
Improving the reliability of commodity operating systems. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003
- [18] A. Ganapathi, V. Ganapathi, D. Patterson
Windows XP kernel crash analysis. Proceedings of the 2006 Large Installation System Administration Conference, 2006
- [19] N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, and Gilles Muller
Faults in linux: ten years later. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11), USA, 2011
- [20] David Wheeler
Secure programmer: Prevent race conditions (2004)
<http://www.ibm.com/developerworks/linux/library/l-sprace/index.html>
- [21] KEDR Manual
http://code.google.com/p/keDr/wiki/keDr_manual_overview
- [22] KEDR wiki: Problems Found
http://code.google.com/p/keDr/wiki/Problems_Found
- [23] Thread Sanitizer Manual
<http://code.google.com/p/thread-sanitizer/w/list>
- [24] Kernel Strider Manual
http://code.google.com/p/kernel-strider/wiki/KernelStrider_Tutorial
- [25] Project: Implement a Lightweight Data Race Detector for Linux Kernel Modules on x86
Google Summer of Code 2012
<http://www.google-melange.com/gsoc/project/google/gsoc2012/nkomarov/7001>
- [26] Linux Kernel Documentation: Kprobes
<http://www.mjmwired.net/kernel/Documentation/kprobes.txt>
- [27] Prasad Krishnan
Hardware Breakpoint (or watchpoint) usage in Linux Kernel. Ottawa Linux Symposium, 2009
<http://kernel.org/doc/ols/2009/ols2009-pages-149-158.pdf>
- [28] Intel 64 and IA-32 Architectures Software Developer Manuals
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

Mobile Learning Systems in Software Engineering Education

Liliya Andreicheva

Institute of Computer Science & Information Technologies
Kazan Federal University
Kazan, Russia
liliya.andreicheva@gmail.com

Rustam Latypov

Institute of Computer Science & Information Technologies
Kazan Federal University
Kazan, Russia
roustam.latypov@kpfu.ru

Abstract— The latest achievements in the information sciences area are mostly connected with the mobile technologies. The significant growth in this area attracts more and more users worldwide. In this paper we concentrate on the mobile learning aspect in education process, specifically in learning management systems. We propose a new design approach which consolidates different front-end representations of the learning system with the single back-end instance. To increase the efficiency of the system we consider new module like preliminary homework checking module, which will be useful especially in software engineering field, and additional statistical and feedback modules.

Keywords— *m-learning; e-learning; mobile technologies; LMS; CMS; learning system; education; software engineering.*

I. INTRODUCTION

In recent years there has been a tremendous growth of mobile technologies worldwide. Nowadays every day the technologies are getting more and more advanced. Modern people use smartphones and tablets almost everywhere – at work, at home, at university, etc. They read newspapers and journals online, check their e-mails while driving back home from work, post in social networks, and play games and chat. According to statistical information [1] the influence of the mobile devices will keep increasing in the nearest future. As we observe most of the users of popular gadgets are young people, students or pupils. This gives us an idea that we can turn mobile technologies into a powerful tool in the educational process.

Obviously, education is one of the most important steps in each young person's life. Unfortunately, not everyone understands it and as the result more effort is needed from the teachers and professors. As mobile technologies cause a lot of interest in the modern society, this fact can be used as an advantage in the educational process. We propose new design principles for the educational systems, whose main idea lies in the unity of back-end and variety of front-end forms. The common server part and the different ways to access information from the client side increases the educational system availability with the minimum set of requirements (basically you need just an Internet access from your mobile device). This paper consists of three parts. In the first part we define the main principles of the e-learning system usage,

especially in the software engineering field. Second part contains some analysis of usage of electronic learning systems in Russia and worldwide. The third part is dedicated to our proposal of new design principles.

II. E-LEARNING AND M-LEARNING

Soon after an expansion of the computer technologies, the e-learning terminology became regular in educational process. Although in recent years as a product of the growth of the mobile industry we received a new term – m-learning. What is m-learning? M-learning states for mobile learning, which suggests the learning process is organized with the usage of mobile devices. This fact makes it available for everyone, who has a smartphone or a tablet. One of the greatest advantages of m-learning is that it is independent from time and place.

The next question is why we need anything else if m-learning systems are available at any time and any place? Each method has its own advantages and disadvantages. The approach is based on the principles of work of different devices. For example, it is much more convenient to read a book from tablet than from smartphone. Although if you want to download this book and use information from it in your report, like different citations and images, you need a good text editor, which is most likely installed on your PC. The main idea is the consolidation of all forms of front-end to obtain the most efficient system for students and professors. These technical specifications define a set of requirements to the system, which we will discuss in the third part of this paper.

One of the drawbacks of the e-learning process is that it is not equally efficient in different areas of study. For example, medical disciplines which require a lot of practical exercises and direct communication with patients cannot be fully automated within the e-learning process. Let us look specifically on the software engineering (SE) area. The area itself is very wide and contains a lot of subjects that should be given to the student. An important aspect that is giving SE a great advantage in the automation of learning systems is that most of the subjects are technical: mathematics, physics, programming sciences, etc. The distinctive characteristics and specifics of this type of sciences allow simplifying the control units significantly. Various tests, quizzes, multiple-choice questions with strict answers can be created easily for any

software engineering course, while for art course you should show your personal vision of the topic and there is generally more than one correct answer. Therefore we propagate a wide usage of e-learning and m-learning technologies in the SE area.

Speaking in terms of international practices, we see that the major universities in the world support a lot of various programs based on the e-learning methods. Statistical analysis shows that every year more and more students enroll in online university services provided by the best universities in the world. This is a great opportunity for students with disabilities, international students and those who cannot afford to pay full tuition for getting the degree on campus. Unfortunately, nowadays leading universities in Russia are just starting to deploy the e-learning systems. Speaking about m-learning, for Russian universities it is still a future project. In my opinion we should start working actively in this area, because now this is a perspective future. We can see the growth of usage of the mobile application that has been deployed on campuses of American universities and colleges in 2010 and 2011 (see “Fig. 1”) [2]. Based on the graph data, we can observe that values have doubled within a single year. Taking into consideration the speed of the development of the mobile technologies, we need to act fast to be able to compete with top universities in the world.

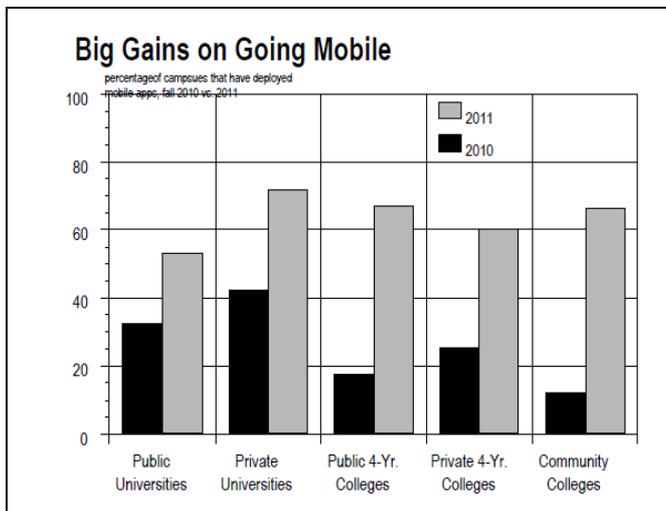


Fig. 1. Big Gains on Going Mobile, percentage of campuses that have deployed mobile apps, fall 2010 vs fall 2011 [2].

III. USAGE OF MOBILE TECHNOLOGIES

According to statistics [3], US, Japan and South Korea are top most consumers of the mobile markets. The forecasted revenue is about \$14 billion by 2014. This number explicitly shows the size of the market and its growth. Although the research results say that more than 22% of the Internet users in cities with population over 100,000 people in Russia access the Web from mobile devices, e-learning and m-learning technologies are not so widespread.

In recent years education became one of the prior areas in the country. In Soviet Union the level of education was one of the highest in the world. The hard political times on the edge of the centuries destroyed the educational system. Nowadays the best universities in the country should work a lot to achieve the

same level that foreign schools are offering. Thus the area of e-learning has become very important. If we look at the way the process is organized abroad we can see that in most of universities there is an internal system for students and professors, which offer a variety of options, from enrollment and scheduling to taking courses online and making presentations online. This is a very important step to every school. In general e-learning process attracts a lot of new students to universities, increases its popularity worldwide and makes education more approachable and affordable for different groups of people. The initial integration of new technologies will definitely require some investments but all the costs will be reimbursed later. For students taking online courses also has some financial benefits. There is a good example the savings the famous company IBM achieved using e-learning practice. In their training process for new managers they used online learning approach and it saved them more than 24 million US dollars. The new technologies used allowed to give 5 times more information and at the same time the cost for one study day was reduced from \$400 to \$135. The other advantage of e-learning approach is the variety of forms that can be integrated in the educational process. The simplest one will be usage of course management and learning management systems (CMS and LMS) in the specific course. More solid approach is the application for the whole set of subjects, which prepare a student for some degree or certification exam. In Russia this is a rare practice, but in our opinion this is the future of the educational process for big universities.

Let us specify some use cases for the e-learning system to obtain the set of requirements. We consider two main roles in the system: professor and student. There are a lot of different aspects that should be taken into consideration in the system, but in the use cases we concentrate on the aspects that are important in the m-learning scope.

One of the regular situations is that professor has to make some changes in the schedule and this information should be sent quickly. Mobile technologies solve this question easily. Various methods can be implemented, via e-mail, via text messages in the message exchange system, but as everybody now has a mobile phone next to him/her all the time, the push messages from the mobile application are the easiest solution. The mechanism of push messages can be used in many cases; therefore there should be a configurable notification system, which will allow providing the most recent information to both professors and students as soon as any change occurred. As we have mentioned before schedule update events are very important, information that new learning materials are available or homework grades are posted. Students are usually mostly interested in their grades and they want to know this information as soon as possible. So it is more convenient for them to check the mobile application rather than to wait till they get back home and log in to the e-learning system on their PC. The specific use of notifications for professors can help to track if all the homework was submitted on time. Another example will be notifications from the administrator of the system about technical issues, like “Servers A and B will be rebooted in 5 minutes. Do not run any tasks at that time”, etc. The advantage of using mobile technologies here is the independence from place and time of the target object of

notification and the speed. Generally viewing message from the mobile app is faster than checking an e-mail or viewing message through the LMS.

Another important aspect, especially for students, is the availability of the materials. Imagine a student, who is working for some company and is getting a degree at the same time. In Russia this is a very widespread situation. The student spends a lot of time in transport, it can be either public transportation, which is usually slow, or traffic jams.

But he does not want to lose this time, he better review once again lectures for the test, which is later the same day. In this case using laptops is not a suitable solution as they are still big and heavy enough. Tablets and smartphones can easily solve the problem, even when you are driving you can enable the voice support on the device, so that it will be reading text to you aloud. By the way, here we should also take into consideration the technical specifications of different gadgets. It is not always convenient to read from you smartphone if the screen is not big enough, but if you have the headset, you can listen to the same lecture. Or you can watch the video recorded during the lecture. Tablets are more user-friendly for these specific purposes. For professors access to homework assignments and tests is more valuable. Thus when they get into the same situation with transportation they can look through the homework tasks or run some automated procedures that check tests and quizzes submitted by the class. These actions can be performed by one button click, they start the job on server and by the time professor gets home he already has a report with the results of the last test. This kind of automation saves a lot of time, which allows professors to spend it later on research and student questions.

These examples show the particular use cases for m-learning, but we have other forms of the front-end, like web-based console and regular PC program. In the next part we will discuss how the whole system should be working together.

IV. SYSTEM DESIGN

The main idea of our proposal is to have the unified concept for different representations of the front-end of the application. This will allow organizing the LMS and CMS in the rational and user-oriented way. The general practice shows that it is much more difficult to upgrade the existing software, which may be absolutely unsuitable for the new needs. Thus a good design approach should specify all the use cases and the corresponding requirements in advance. This is one of the reasons why the idea of development of new system is better than using some existing ones. An individual project can be turned up for the specific goals, while any modification of an existing tool presented on the market is a long time- and labor-consuming process. The other reason is that most of the existing systems are expensive. Although there is an open-source segment of the market, if we look at the theory of the delayed expenses this approach becomes inapplicable for large educational organizations like universities and colleges. At some point there is always a risk that developers of open-source software will decide to stop the support of this particular product that you are using. This can cause a lot of problems for the users, as eventually it leads not only to the change of an

existing system but also to the migration of the information from one system to another, which is also expensive and inconvenient.

Although we consider three types of representation of the user interface for the system (see “Fig. 2”), we concentrate mainly on the mobile application and PC programs. The reason for our choice is that these two parts have different purposes. As we have mentioned earlier the use cases for the m-learning application first of all allow the user to be independent from time and location. The set of features for this app will be defined mostly by this factor, however we also take into account that some actions are inconvenient to perform from the mobile device (like file upload, download, text/graph editing, submission of programming assignments, etc.). For the PC we consider a complete system with support of a variety of features for students, professors and administrators. As we have mentioned several roles above, let us elaborate on that.

We have already mentioned students and professors as the target users of the application. Besides them there is one more user role – administrator, who is responsible for registration of other users, helping them with any questions about system usage and controlling the whole system. In general we suppose that system includes plenty of courses from various departments. To handle the department information accurately and to be able to control the department schedules we propose a subgroup of administrators called department administrators. Basically the administrator can create another administrator-user who will be able to work only with data from his department (see “Fig. 3”). One of the possible scenarios of system development is integration with some other useful services of university campus, for example, information about operation hours of departments, library, on-campus cafes, etc. In this case the role of an administrator will be to include handling information for these additional services. We will not discuss the administrator role a lot because he is basically only using the PC application, while in this paper we concentrate on the m-learning aspect.

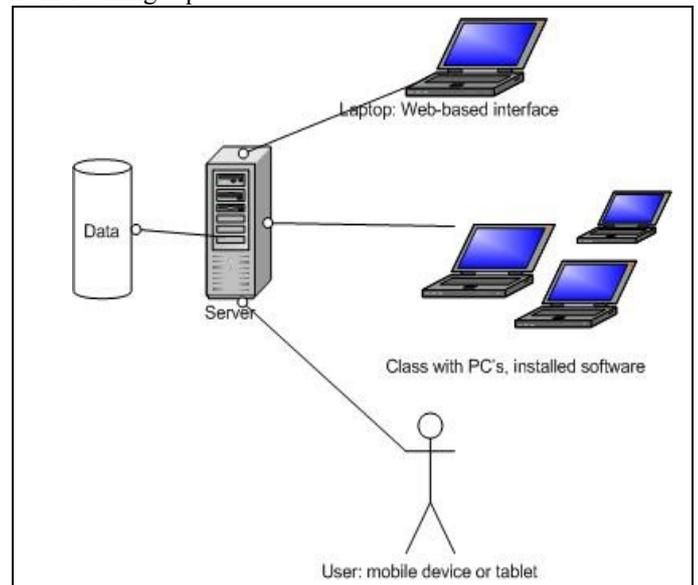


Fig. 2. The organization of the system.

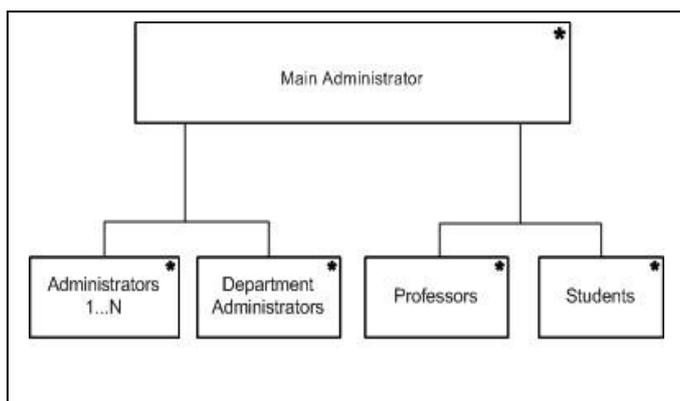


Fig. 3. Hierarchy of roles in the system.

Based on the use cases that we have mentioned above the primary purpose of the mobile application is to keep the user updated with the latest progress that is going on in the system. Thus we should concentrate on the notification system and monitoring console. From our point of view notification mechanism is more important for students, while monitoring is more valuable for professors. Nevertheless in the design we consider both types of users being able to use these components.

Generally notifications can be divided into two types: created by system and created by user. Each of these types of messages includes the following levels: information, warning, error. For better user experience we also suggest three types of importance of every message (low, regular, high). This makes system more flexible, because depending on the user's approach he can configure the individual preferences. The importance level will help the user handle urgent questions with high priority faster. On the other hand, from the system development point of view, such notification mechanism will help improve the product stability and troubleshooting. The system notifications are configured by administrator. They are designed mainly for administrators to keep the system maintained. But regular users should also be aware of the current situation. For example, when student is trying to upload his homework but the server is rebooting, he should get an appropriate alert to be able to perform this operation later. This will be a user-friendly behavior and will decrease the number of errors. The user should be able to set up notification mechanism in general and for any particular course. The most important issues from the system point of view include system failures, database failures, server connection problems, protocol problems, running jobs problems (running automated homework or quiz checking, etc.), user access problems, etc. The alert message should contain description of the problem, error code if applicable, and suggestions of possible solution. The user defined notifications include information about any changes in the course schedule, homework assignments, lecture materials, forum updates, tests and examination results, course announcements, etc. The set of these messages should be the same for students and professors. As it might be the case where one course is divided between two professors and both have access to course materials, so one should be notified of what another is changing in the course.

Monitoring is a great tool mostly for professors. Besides some general statistical information about popularity of the course, individual student performance, overall student performance, etc., provided logical set of rules will be able to analyze performance of students in general and individually within the course and give some recommendations. For example, module A was successfully learned by 90% of students, 5% need to review the module once again, 5% finished the module with average knowledge of the subject. This information should be interesting to both teachers and students. From the development point of view it will require implementation of some analytical algorithms on the back-end. More complex approach can involve some expert systems or self-learning neural networks. Monitoring of individual performance is also very important for students. We should remember that one of the main characteristics of the e-learning process is individual factor. The whole concept of e-learning supposes that most of work is done individually and professor is controlling it remotely. This requires good self-organizational abilities, time management and motivation. Thus the overall performance of the student should be given in comparison to the class that will increase student's motivation.

One of the most important components of educational process is control of the level of knowledge obtained, thus we consider in e-learning systems particular attention should be paid to this aspect. Information technologies brought automation to the regular slow processes and this greatly increased the performance in various areas. Therefore, the higher is the level of automation of the system, the easier and more practical it is in use. Our idea supposes adding some extra control on the stage of homework submission. We offer a module which will execute some preliminary control. The goal of this module is to have some self-controlling tool for the students. The main idea of this approach is that the system performs some additional checks when uploading homework. The result is an informational message demonstrating the validity of the submitted task. Based on the response of the system student can then re-upload his homework with any modifications needed, if the original exercises have been carried out not quite right. To implement this approach, a set of tests to verify the job is needed. Such set can have quite different views. From basic check for the correct answers of the math problems to various tests for the correctness of the output of some programs delivered as assignments for programming courses. This module will depend a lot on the particular subject and the approach of the professor. This tool will be easy to configure for the various software engineering courses. Overall the effort for making this module work properly in technical courses is less, than it would be for art courses. Nevertheless we consider this approach useful in every area. Basically the efficiency of the module depends on the professor. For programming courses, for example, the control condition to the minimum practical tasks may be simply an ability to compile the program, and more serious approach will have some test units, checking the correctness of the input parameters, output results and test the operation of the program on different operating systems, etc. From the students point of view this module is very practical and provides additional help. Obviously with proper organization of the time they can upload the homework in advance to the system and

get a preliminary result. Now they know whether their work fits the minimum requirement for this task. Such verification system does not involve full check of the assignment on-a-fly, but allows students to significantly improve their results. For professors module also provides an effective time management: time taken to prepare the preliminary tests for jobs will reduce the time needed to check homework, and improve the quality of the provided solutions.

Nowadays one of the common methods of improving something is polling the audience. Thus in many areas system of feedback and suggestions is used. From our point of view, the presence of such a system will make the e-learning system more effective. The use of such a module in e-learning systems is simple for users, thanks to the process automation; the results are calculated quickly and immediately available for the professor. This module performs several functions:

- Evaluation of the professor - how effective his method of teaching is, whether he explains the material good enough, whether he answers the questions and emails quickly, etc.
- Course evaluation - whether course content suites expectations, do selected materials cover the subject, were the home assignments effective, how is the control system, how popular is this course among students, etc.

The big advantage of this survey is that it may be configured from the same set of questions regardless of the course topics, and include some specific items that will be important for a particular teacher and course. If there is no such a need for the specific questions, the section where comments allowed can help a lot. The generic system of evaluation of courses and instructors can be organized when using same questions in the surveys. This approach allows creating a rating system. Rating will help students to select from a variety of courses, instructors. For the organizations, that use e-learning system for professional trainings rating system will help to evaluate the effectiveness of professors and courses.

From the developer's point of view an e-learning system is a complex application primarily divided into front-end and back-end. In our case we are basically talking about a client-server architecture where the client side has different views. On the one hand, the client is a separate application for the PC, on the other hand, it is a mobile application, which uses a completely different design principles. However, both of them are parts of the same system, working with a server, which stores all the data. As we have mentioned earlier we concentrate primarily on the mobile and PC client views, but as another view we consider a web console. Running an e-learning system from the browser also requires some other technologies. At this point an important issue is to keep the

client-server protocol the same for communication between all parts of the application. One of the other important questions is synchronization. When using file-sharing service Dropbox, files are downloaded from one device to the shared folder, and then are displayed at all devices that run with the appropriate application. The same idea should be used in the proposed system. So before actual implementation is done, a careful study of the mathematical model technologies should be done, because every part of the application is very different.

In this paper we present the system design. The initial prototype is under construction now. However, the described approach presents fully the main features of the whole system. The heart of the system is the server part, which takes most of the development time. Overall, development process contains a lot of pitfalls, which are connected with different problems – from the technologies point of view and

The novelty of the solution lies not only in the choice of a new long-term approach to client-server architecture, but also in the introduction of the new modules for the convenience of teachers and students. Our design approach is promising in terms of end-user response, as it is a priority to increase the availability and convenience of e-learning systems for users. Based on statistical data [1], we conclude that the audience of Internet users and advanced mobile devices is increasing. Education has always been a key element in the development of the society. Thus, the introduction of the latest technology innovations in the educational process can only increase the development of the state and society.

REFERENCES

- [1] Blinov, D. (2012). Statistics of usage of mobile devices, platforms and applications. Retrieved from: <http://beamteam.ru/2012/09/mobile-platforms-share-2012/>
- [2] A Profile of the LMS Market (page 18), CampusComputing, 2011. Retrieved from: http://www.campuscomputing.net/sites/www.campuscomputing.net/files/Green-CampusComputing2011_4.pdf
- [3] Nicole Fougere, US Leads the Global Mobile Learning Market , 2010 <http://www.litmos.com/mobile-learning/us-leads-the-global-mobile-learning-market-mlearning/>
- [4] Kerschenbaum, Steven (04). "LMS Selection Best Practices" (White paper). Adayana Chief Technology Officer. pp. 1–15, 13 February 2013. Retrieved from: http://www.trainingindustry.com/media/2068137/lmsselection_full.pdf
- [5] Ellis, Ryann K. (2009), Field Guide to Learning Management Systems, ASTD Learning Circuits. Retrieved from: http://www.astd.org/~media/Files/Publications/LMS_fieldguide_20091
- [6] Wikipedia. <http://www.wikipedia.org/>

Hide and seek: worms digging at the Internet backbones and edges

Svetlana Gaivoronski
Computational Mathematics and Cybernetics dept.
Moscow State University
Moscow, Russia
Email: sadie@lvk.cs.msu.su

Dennis Gamayunov
Computational Mathematics and Cybernetics dept.
Moscow State University
Moscow, Russia
Email: gamajun@cs.msu.su

Abstract—The problem of malicious shellcode detection in high-speed network channels is a significant part of the more general problem of botnet propagation detection and filtering. Many of the modern botnets use remotely exploitable vulnerabilities in popular networking software for automatic propagation. We formulate the problem of shellcode detection in network flow in terms of formal theory of heuristics combination, where a set of detectors are used to recognize specific shellcode features and each of the detectors has its own characteristics of shellcode space coverage, false negative and false positive rates and computational complexity. Since the set of detectors and their quality is the key to the problem's solution, we will provide a survey of existing shellcode detection methods, including static, dynamic, abstract execution and hybrid, giving an estimation to the quality of the characteristics for each of the methods.

Keywords-shellcode; malware; polymorphism; metamorphism; botnet detection;

I. INTRODUCTION

Since the early 2000's and until the present time botnets are one of the key instruments used by cybercriminals for all kinds of malicious activity: stealing users' financial information, bank accounts credentials, organizing DDoS attacks, e-mail spam, malware hosting et cetera. Among the recent botnet activity we could mention the Torpig botnet, which was deeply investigated by the UCSB research group Torpig, the Zeus botnet involved in FBI' investigations which ended in arrest of over twenty people in September 2010 [6], and also the Kido/Conficker botnet, which has attracted the attention of security researchers since the end of 2008 and is still one of the most widespread trojan programs found on end users computers [4].

Despite of the fact that malware tends to propagate via web applications vulnerabilities, drive-by downloads, rogue AV software and infecting legitimate websites more often, the significance of remotely exploitable vulnerabilities in widespread networking software does not seem to have faded out in the following years, since the large installation base of the vulnerable program warrants very high infection rates in case of the zero-day attacks. Besides, drive-by downloads often make use of remotely exploitable vulnerabilities in the client software like Microsoft's Internet Explorer, Adobe Reader or Adobe Flash. A typical remotely exploitable

vulnerability is a kind of memory corruption error - heap or stack overflows, access to the previously freed memory and other overflow vulnerabilities. Modern malware utilizes so called "exploit packs", commercially distributed suites of shellcodes for many different vulnerabilities, some of which may be unknown to the public. For example, the Conficker worm exploited several attack vectors for propagation: the MS08-67 vulnerability in Microsoft RPC service, dictionary attack for local NetBIOS shares and propagation via USB sticks autorun. Nevertheless, among all these propagation methods exploitation of the vulnerabilities in the networking software gives the attacker (or the worm) the best timing characteristics for botnet growth, because it requires no user interaction.

We could conventionally designate the following main stages of the botnets life cycle: propagation, privilege escalation on the infected computer, downloading trojan payload, linking to the botnet, executing commands from the botnet's C&C, removal from the botnet. Comparing the ease of botnet activity detection and differentiating it from normal Internet users activity, the propagation stage would be the most interesting as it involves computer attack, which is always an anomaly. The stages that follow successful infection - trojan extensions downloads, linking to botnet and receiving commands are usually made using ordinary application level protocols like HTTP or (rarely) IRC, different variations of P2P protocols, so that these communications are fairly easy to render to look like normal traffic. At the same time the propagation stage almost always involves shellcode transfer between attacker and victim, therefore it is easier to detect than other stages. This is why memory corruption attacks and their detection are important for modern Internet security.

A. Shellcodes and memory corruption attacks

A memory corruption error occurs when some code within the program writes more data to the memory, than the size of the previously allocated memory, or overwrites some internal data structures like malloc() memory chunks delimiters. One typical example of a memory corruption attack is stack overflow, where the attacker aims at overwriting the function return address with an address somewhere within

Activator	Decryption routine	Shellcode payload	Return address zone
-----------	--------------------	-------------------	---------------------

Figure 1: Example of possible shellcode structure. Activator may be NOP-sled or GetPC code or alike.

the shellcode. Another example of a memory corruption attack is a heap overflow which exploits dynamic memory allocation/deallocation scheme in the operating system's standard library.

An example of a possible shellcode structure is shown at figure 1. Conditionally, we could break shellcodes into classes depending on which special regions they contain, where each shellcode region carries out some specific shellcode function, including detection evasion. For example, these could be regions of NOP-equivalent instructions (NOP-sled) or GetPC code as an activator, a decryption routine region for encrypted shellcodes, shellcode payload or return address zone.

In terms of classification theory we could define a shellcode as a set of continuous regions of executable instructions of the given architecture, where regions are associated by the control flow (following each other sequentially or linked to each other with control flow transfer instructions), and where one or more shellcode features are present simultaneously (i.e. it contains an activator, decryptor, shellcode payload zone or return address zone, associated by control flow).

There are significant numbers of existing and ongoing research activities which try to solve shellcode detection in network flow problem. These methods can be grouped into classes in two ways - by the type of analysis they perform (static, dynamic, abstract execution, hybrid) or by the types of shellcode features they are designed to detect (for example, activator, decryptor, shellcode payload, return address zone). An important observation is that most modern research papers are focused on IA32 (EM64-T) architecture, since most Internet-connected devices running Windows platform use this architecture and, besides, some of Intel Architecture instruction set features make memory corruption exploitation easier. This may change in the following decade when the broadband wireless connections for mobile devices become more common.

B. Computation complexity problem

Since we primarily aim at detecting network worms propagation (botnet growth) and not just remote exploitation of memory corruption vulnerabilities, our task has several certain peculiarities. Like any massive phenomenon worm propagation is best monitored in large scale, than at the end point of the attacked computer. This means that we should better try to detect worm propagation analyzing network data in transit at the Tier-2 channels or even Tier-1 channels. And in this case we inevitably fail because of the lack of computational power. There are two famous empiric laws which reflect the evolution of computation and

computer networks - these are Moore's law and Gilder's law. Moore's law states that the processing power of a computer system available for the same price doubles every 18 months, and the Gilder's law says that the total bandwidth of communication systems triples every twelve months (see figure 2). The computational power of a typical computer system available for network channel analysis tends to grow slower than the throughput of the channel. The real-time restrictions for filtering devices also become more strict. For example, the worst case scenario for 1Gbps channel which is a flow of 64-byte IP packets at the maximum throughput gives us about 600ns average time for each packet analysis if we want to achieve wire-speed, and it gives only about 60ns in case of 10Gbps channel. This trend makes requirements for computational complexity of the algorithms utilized by network security devices more severe each year. That's why algorithms used for inline shellcode mitigation should have reasonable computational complexity and allow implementation in the custom hardware (FPGA, ASIC).

We also should not forget that backbone network channels like those connecting two or more different autonomous systems are especially sensitive to the false positives of the filtering device, because they lead to denial of service for the legitimate users.

In this paper we formulate the task of the malicious shellcode detection in the high-speed network channels as a multi-criteria optimization problem: how to build a shellcode classifier topology using a given set of simple shellcode feature classifiers, where each simple classifier is capable of detecting one or more simple shellcode features with zero false negative rates, given computational complexity and false positive rates within its shellcode classes, so that to provide the optimum aggregate false positive rates along with computational complexity. The key element of any solution of this task is the set of simple classifiers. Thus, we provide a survey of the existing methods and algorithms of shellcode detection, which could be used as simple classifiers for the aggregate detector. In this survey we pay special attention to the class coverage, false positives rates and computational complexity of each method or algorithm. The structure of this paper is as follows. In the second section the classes of shellcode features are given and the main part of the section is the shellcode detection methods survey. In the third section we provide estimations of the key methods characteristics, which are essential for solving multi-criteria optimization problems. In the final two sections we discuss the results of the survey and suggest the formal task definition for building hybrid classifier as a oriented filtering graph of simple classifiers with optimal

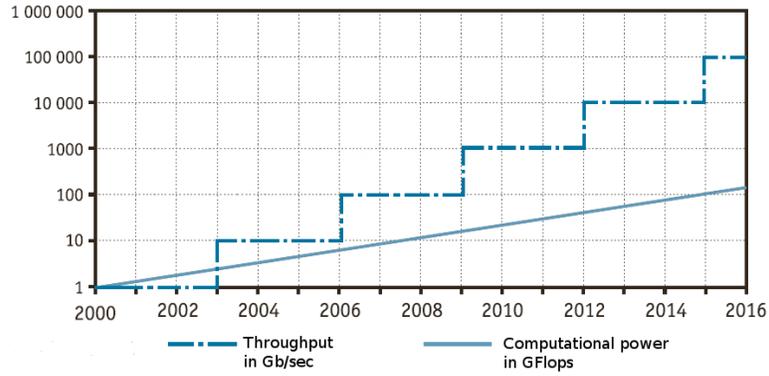


Figure 2: Moore and Gilder laws - the network channels throughput leaves the computational power behind.

computational complexity and false positive rates.

II. SHELLCODE DETECTION METHODS

This section provides a classification of malicious objects and methods of shellcode detection. In addition, we will give a description of existing methods. For each method, we will briefly describe the basic idea. We will also describe classes of shellcode and their coverage, false positive rates and, where possible, we will give the computational complexity for the methods.

Let $S = \{Seq_1, \dots, Seq_r\}$ be a given set of sequences of executable instructions, later referred to as *object* S . We assume that all instructions in the object are valid instructions of the target processor. Let us define several definitions for S , using terminology from [1].

Let us consider a set of *features* $Mal = \{m_1, \dots, m_n\}$ of malicious instruction set (a malicious object) and a set of *features* $Leg = \{l_1, \dots, l_k\}$ of a legitimate set of instructions.

Suppose we are given a set M of *malicious* objects. Set M is covered by a finite number of subsets K_1, \dots, K_l :

$$M = \bigcup_{i=1}^l K_i.$$

Subset $K_j, j = \overline{1, l}$ is called *the class* of malware. Each class K_j is associated to the set of features $Mal(K_j)$ and $Leg(K_j)$ from the set of malicious features Mal and legitimate features Leg respectively. In addition, the partition of M to K_j classes conducted in a way that

$$Mal = \bigcup_{i=1}^l Mal(K_i)$$

and

$$Leg \neq \bigcup_{i=1}^l Leg(K_i)$$

in general.

Each class K_j is assigned with *elementary predicate*

$$P_j(S) = (S \in K_j), P_j(S) \in \{0, 1, \Delta\}$$

(object $S \in K_j$; $S \notin K_j$; unknown). The information about the occurrence of object in the class K_1, \dots, K_l is encoded by vector $(\alpha_1 \alpha_2 \dots \alpha_l)$, $\alpha_i \in \{0, 1, \Delta\}$, $i = \overline{1, l}$.

Definition 1: Instruction set S is called a legitimate, if its information vector is null $|\tilde{\alpha}(S)| = 0$. In other words, the object is considered as legitimate iff it is not contained in any of the classes K_j of malicious set M .

Definition 2: Instruction set S is called malicious if the length of its information vector is equal to or greater than 1: $|\tilde{\alpha}(S)| \geq 1$. In other words, the object is considered as shellcode if it is contained at least in one of the classes K_j of malicious set M .

The problem of detecting malicious executable instructions is to calculate the values of the predicates $P_j(S) = (S \in K_j)$ and to construct information vector $\tilde{\alpha}^A(S)$, where A is the detection algorithm.

Definition 3: False negatives FN of algorithm A is the probability that the information vector of S resulted by algorithm A is null, but the veritable vector of object S is not null.

$$FN(A) = \mathbf{P}(|\tilde{\alpha}^A(S)| = 0 \mid |\tilde{\alpha}(S)| \geq 1), S \in M.$$

In other words, it is probability that a malicious object is not assigned to any of the classes K_j of malicious set M .

Definition 4: False positives FP of algorithm A is the probability that the length of information vector of object S returned by algorithm A is greater than or equal to 1, but the veritable vector of object S is null.

$$FP(A) = \mathbf{P}(|\tilde{\alpha}^A(S)| \geq 1 \mid |\tilde{\alpha}(S)| = 0), S \notin M.$$

In other words, it is the probability of classifying a legitimate object to at least one of the classes K_j of malicious set M .

A. Shellcode features classification

As previously mentioned, the entire set of malicious objects M is covered by the classes K_1, \dots, K_l : $M = \bigcup_{i=1}^l K_i$. Let us define the classes K_1, \dots, K_l with respect to the structure of malicious code. Thus, the set M can be classified as follows:

1) Activators:

- K_{NOP_1} - class of objects containing simple NOP-sled - a sequence of nop (0x90) instructions ;
- K_{NOP_2} - objects containing one-byte NOP-equivalents sled;
- K_{NOP_3} - objects containing multi-byte NOP-equivalents sled;
- K_{NOP_4} - objects containing four-byte aligned sled;
- K_{NOP_5} - objects containing trampoline sled;
- K_{NOP_6} - objects containing obfuscated trampoline sled;
- K_{NOP_7} - objects containing static analysis resistant sled;
- K_{GetPC} - objects containing GetPC code.

2) Decryptors:

- K_{SELF_UNP} - self-unpacking shellcode class;
- K_{SELF_CIPH} - self-deciphering shellcode class.

3) Payload:

- K_{SH} - non-obfuscated shellcode class;
- K_{DATA} - class of shellcode with data obfuscation. For example, ASCII character set can be replaced by UNICODE;
- K_{ALT_OP} - class of shellcode obfuscated by the insertion of alternative operators;
- K_R - class of shellcode, obfuscated by instruction reordering in the code;
- K_{ALT_I} - class of shellcode, obfuscated by replacing the instructions with instructions with the same operational semantics;
- K_{INJ} - class of shellcode, obfuscated by code injection;
- K_{MET} - class of metamorphic shellcode - shellcode whose body is changing with respect to semantic structure maintaining;
- K_{NSC} - (non-self-contained) - class of polymorphic shellcode which does not rely on any form of GetPC code, and does not read its own memory addresses during the decryption process.

4) Return address zone:

- K_{RET} - class of shellcode which can be detected by searching for the return address zone;
- K_{RET+} - class of shellcode whose return address is obfuscated. For example, one can change the order of lower address bits. In this case, the control will be transferred to different positions of the stack, but always

in any part of NOP-sled. In this case, the functionality of the exploit will not be compromised.

B. Methods classification

According to the principles at work, shellcode detection methods can be divided into the following classes:

- *static methods* - methods of code analysis without executing it;
- *abstract execution* - analysis of code modifications and accessibility of certain blocks of the code without a real execution. The analysis uses assumptions on the ranges of input data and variables that can affect the flow of execution;
- *dynamic methods* - methods that analyze the code during its execution;
- *hybrid methods* - methods that use a combination of static and dynamic analysis and the method of abstract interpretation.

From a theoretical point of view, static analysis can completely cover the entire code of the program and consider all possible objects S , generated from the input stream. In addition, static analysis is usually faster than dynamic. Nevertheless, it has several shortcomings:

- A large number of tasks which rely on the program's behavior and properties, can't be solved by using static analysis in general. In particular, the following theorems have been proved in the work of E. Filiol [13]:
Theorem 1: Problem of detecting metamorphic shellcode by static analysis is undecidable.
Theorem 2: The problem of detection of polymorphic shellcode is NP-complete in the general case.
- The attacker has the ability to create malicious code which is static analysis resistant. In particular, one can use various techniques of code obfuscation, indirect addressing, self-modifying code techniques, etc.

In contrast to static methods, dynamic methods are resistant to the code obfuscation and to the various anti-static analysis techniques (including self-modification). Nevertheless, the dynamic methods also have several shortcomings:

- they require much more overheads than static analysis methods. In particular, a sufficiently long chain of instructions can be required to conclude whether the program has malicious behavior or not;
- the coverage of the program is not complete: the dynamic methods consider only a few possible variants of program execution. Moreover, many significant variants of program execution can not be detected;
- the environment emulation in which the program exhibits its malicious behavior is difficult;
- there are detection techniques for program execution in a virtual environment. In this case, the program has the ability to change its behavior in order not to exhibit the malicious properties.

C. Static methods

A traditional approach for static network-based intrusion detection is signature matching, where the signature is a set of strings or regular expression. Signature based designs compare their input to known, hostile scenarios. They have the significant drawback of failing to detect variations of known attacks or entirely new intrusions. Signatures themselves can be divided into two categories: context-dependent and signatures that verify the behavior of the program.

One example of signature-based methods is **Buttercup** [12] - a static method that focuses on the search of the return address zone. The algorithm solution is simply to identify the ranges of the possible return memory addresses for existing buffer-overflow vulnerabilities and to check the values that lie in the fixed range of addresses. The algorithm considers the input stream, divided into blocks of 32 bits. The value of each byte in the block is compared with the ranges of addresses from the signatures. If the byte value falls into one of the intervals, an object S is considered as malicious. Formally,

$$|\tilde{\alpha}^{BUTTERCUP}(S)| \neq 0 \Leftrightarrow \exists I_j \in S : val(I_j) \in [LOWER, UPPER],$$

where $LOWER$ and $UPPER$ - lower and upper limits of the calculated interval, respectively. In the notions of introduced model, we assume that the second part of the expression is predicate $P_j(S)$, defining membership of an object S to one of the classes of malware. Since this method relies on known return addresses used in popular exploits, it becomes unusable when the target host utilizes address space layout randomization (ASLR). Static return addresses are rarely used in real-world exploits nowadays.

Another example of the signature-based methods is the **Hamsa** [14] - static method that constructs context-dependent signatures with respect to a malware training sample. The algorithm selects the set

$$\{S_i \mid \tilde{\alpha}_j(S_i) \neq \{0, \Delta\}\}$$

from the training information. Then the algorithm constructs a signature $Sig_j = \{T_1, \dots, T_k\}$ from that set. The signature itself is a set of tokens $T_j = \{I_{j_1}, \dots, I_{j_n}\}$, where I_{j_i} are instruction. In general, in [14] the following theorem is represented:

Theorem 3: The problem of constructing a signature Sig with respect to the parameter $\rho < 1$ such that $FP(Sig) \leq \rho$ is NP-hard.

The authors make the following assumptions in the problem: let the parameters $k^*, u(1), \dots, u(k^*)$ characterize a signature. Then, the token t added to the signature during signature generation iff

$$FP(Sig \cup \{t\}) \leq u(i).$$

When the signature is generated, the algorithm checks whether it matches to object S or not:

$$|\tilde{\alpha}^{HAMSA}(S)| \neq 0 \Leftrightarrow Sig_j \in S.$$

Another considered static signature-based method is **Polygraph** [15]. The approach builds context-dependent signatures. The algorithm takes different versions of the same object S' as a training set of objects S_1, \dots, S_m for training information. Versions of S are generated by applying the operation of polymorphic changes for m times. With respect to learning information the Polygraph builds three types of signatures. If any of these signatures matches object S then S is considered as malware. The types of signatures are following: i) conjunction signatures Sig_C (consist of a set of tokens, and match a payload if all tokens in the set are found in it, in any order); ii) token-subsequence signatures Sig_{SUB} (consist of an ordered set of tokens); iii) Bayes signatures $Sig_B = \{(T_{B_1}, M_1), \dots, (T_{B_r}, M_r)\}$ (consist of a set of tokens, each of which is associated with a score, and an overall threshold). We define the following predicates:

$$P_C(S) = (Sig \in S) -$$

predicate checks whether objects S matches to conjunction signature;

$$P_{SUB}(S) = (\forall i, j, m, n, k, t : T_{SUB_i} = \{I_m, \dots, I_n\}, T_{SUB_j} = \{I_k, \dots, I_t\} : i < j \Rightarrow m < n) -$$

predicate checks if set of tokens in the objects is ordered;

$$P_B(S) = (\forall i : |T_{B_i}| \geq M_i) -$$

predicate checks whether token exceeds threshold. Then the algorithm can be formally described as:

$$\begin{aligned} |\tilde{\alpha}^{POLYGRAPH^C}(S)| \neq 0 &\Leftrightarrow P_C(S), \\ |\tilde{\alpha}^{POLYGRAPH^{SUB}}(S)| \neq 0 &\Leftrightarrow P_C(S) \& P_{SUB}(S), \\ |\tilde{\alpha}^{POLYGRAPH^B}(S)| \neq 0 &\Leftrightarrow P_B(S). \end{aligned}$$

Among the methods of static analysis, which generating the signature of program behavior, we have considered the method of **structural analysis** [9]. Let us call it Structural in the rest of paper. By training on a sample of malicious objects S_1, \dots, S_m the approach constructs a signature base of program behavior. The object S is considered as malware if it matches any signature in the base. The method checks whether object matches a signature contained in the base by following steps:

- program structure is identified by analyzing the control flow graph (CFG);
- program objects are identified by CFG coloring technique;
- for each signature and for each built program structure the approach analyses, whether they are polymorphic modifications of each other.

Nevertheless, a simple comparison of the control flow graphs is ineffective due to the fact that this isn't robust to the simplest modifications. The authors of the method propose the following modification: subgraphs containing

k vertices are identified. Identification of the subgraph is carried out as follows:

- first, the adjacency matrix is built. The adjacency matrix of a graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (v_i, v_j) according to whether there is an edge from v_i to v_j or not;
- second, a single fingerprint is produced by concatenating the rows of the matrix;
- additionally, the calculation of fingerprints extended to account for colors of vertices (graph is colored according to the type of verticle). This is done by first appending the (numerical representation of the) color of a node to its corresponding row in the adjacency matrix.

Definition 5: Two subgraphs are related if they are isomorphic and their corresponding vertices are colored the same.

Definition 6: Two control flow graphs are related if they contain related K -subgraphs (subgraph containing k vertices). It is believed that if $CFG(S)$ are related to any control flow graph of a malicious object, then the object S itself is malicious.

Let $\{ID\}$ be the set of subgraphs identifiers. Subgraphs contained in the malicious objects from the training data. We define the predicate

$$P_{ST} = id(S) \in \{ID\}.$$

Thus,

$$|\tilde{\alpha}^{Structural}(S)| \neq 0 \Leftrightarrow P_{ST}(S).$$

The **Stride** [10] algorithm is NOP-sled detection method. STRIDE is given some input data, such as a URL, and searches each and every position of the data to find a sled. STRIDE can be formally described as follows: it forms object S from the input stream by disassembling, starting at offset $i + j$ of the input data, for all $j \in \{0, \dots, n - 1\}$. It is believed that the input stream contains a NOP-sled of length n :

$$|\tilde{\alpha}^{STRIDE(n)}(S)| \neq 0,$$

if the object $S = \{I_1, \dots, I_k\}$ satisfies the following conditions:

- $k \geq n$;
- $\exists i : \forall j : j = i, \dots, i + n \Rightarrow (I_j \neq Privileged) \parallel (\exists k : i \leq k < j \ I_k = JMP)$

In other words, it is believed that a sled of length n starts at position i if it is reliably disassembled from each and every offset $i + j$, $j \in \{0, \dots, n - 1\}$ (or from each of the 4th byte) and in any subsequence of S privileged instruction isn't met (or a jump instruction is encountered along the way).

There is an algorithm **Racewalk** [11] which improves performance of the algorithm STRIDE through the decoded

instructions caching. Moreover, Racewalk uses pruning techniques of instructions that are not a valid NOP-sled (for example, if we meet an invalid or a privileged instruction at some position h , it is obvious that the run from offset $j = h\%4$ is invalid. Consequently, the object S formed from the offset $j = h\%4$ will not appear in any of the classes $K_{NOP_1}, \dots, K_{NOP_7}$). Racewalk also uses the instruction prefix tree construction to optimize the process of disassembling.

Styx [8] is a static analysis method, based on CFG analyzing. Object S is believed to be malware if sliced CFG contains cycles. Cycles in the sliced CFG indicate the polymorphic behavior of the object. For such an object the signature is generated in order to use it in its signature base.

Given the object S algorithm builds a control flow graph (CFG). The vertices are the blocks of instruction chains. Such blocks do not contain any transitions. The edges are the corresponding transitions between the blocks. All the blocks in the graph can be divided into three classes:

- valid (the branch instruction at the end of the block has a valid branch target);
- invalid (the branch target is invalid);
- unknown (the branch target is unknown).

Styx constructs a sliced GFG from control flow graph. All invalid blocks and blocks to which invalid ones have the transitions are removed from sliced CFG. Some of the blocks are excluded as well using the technique of data flow analysis, described in [16]. From sliced CFG Styx constructs a set of all possible execution chains of instructions. Next, method considers each of chains to check whether it contains cycles or not. To formalize the algorithm we describe the following predicates. Let $SIG = \{Sig_1, \dots, Sig_n\}$ be the signatures base which constructed from training information. Thus,

$$P_{SIG}(S) = \exists i : (Sig_i \in SIG) \ \& \ Sig_i \subset S$$

is the predicate which verifies that the object matches to one of the previously generated signatures. $P_{cycle}(S)$ is the predicate which checks sliced CFG of S for cycles. Consequently,

$$|\tilde{\alpha}^{STYX}(S)| \neq 0 \Leftrightarrow P_{SIG}(S) \parallel P_{cycle}(S).$$

In contrast to this algorithm, a method **SigFree** [17] statically analyses not CFG but an instruction flow graph (IFG). Vertices of CFG contain blocks of instruction while IFG vertices contain instructions only. An object is considered as malware if its behavior conforms to the behavior of real programs, rather than a random set of instructions. Such heuristic restricts applicability of method on the channels such that the profile of the traffic allows for the transfer of executable programs.

Definition 7: An instruction flow graph (IFG) is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, v_j) \in E$

corresponds to a possible transfer of control from instruction v_i to instruction v_j .

The analysis is based on the assumption that a legitimate object S , consisting of instructions encountered in the input stream, can not be a fragment of a real program. Real programs are assigned with two important properties:

- 1) The program has specific characteristics that are induced by the operating system on which it is running, for example calls to the operating system or kernel library. A random instruction sequence does not carry this kind of characteristics.
- 2) The program has a number of useful instructions which affects the results of the execution path.

With respect to these properties, the method provides two schemes of *IFG* analysis. In the first scheme *SigFree* based on training information constructs a set $\{TEMPLE\}$ of instructions call templates. Then algorithm checks whether object S satisfies these patterns or not. Let us describe the predicate

$$P_1 = \exists t \in \{TEMPLE\} : t \in IFG(S)$$

which checks if *IFG* of S satisfies to any of the templates. Thus,

$$|\tilde{\alpha}^{SigFree_1}(S)| \neq 0 \Leftrightarrow P_1(S).$$

The second scheme is based on an analysis of the data stream. In this scheme, each variable can be mapped from the set

$$Q = \{U, D, R, DD, UR, DU\},$$

where the six possible states of the variables are defined as following. State U : undefined; state D : defined but not referenced; state R : defined and referenced; state DD : abnormal state define-define; state UR : abnormal state undefine-reference; and state DU : abnormal state define-undefine. *SigFree* constructs for an object S state variables diagram - an automaton

$$DSV = (Q, \Sigma, \delta, q_0, F),$$

where Σ is the alphabet, consisting of instruction of object S , and $q_0 = U$ is the initial state. If there is the transition to the final (abnormal state) when parsing S , it is believed that the instruction is useless. All useless instructions are excluded from the object S , resulting in an object $S' \subset S$. Let us describe the following predicate:

$$P_2(S) = |S'| > K,$$

where K - threshold. Thus,

$$|\tilde{\alpha}^{SigFree_2}(S)| \neq 0 \Leftrightarrow P_2(S).$$

There is an algorithm **STILL** [18] which improves the method *SigFree*. The method based on techniques to detect self-modifying and indirect jump exploit code are called static taint analysis and initialization analysis. The method is

based on the assumption that self-modifying code and code using the indirect jump, must obtain an absolute address of the exploit payload. With respect of this the method searches subset $S' \in S$ which obtains the absolute address of the payload at runtime. The variable that records the absolute address is marked as *tainted*. The method uses the static taint analysis approach to track the tainted values and detect whether tainted data are used in the ways that could indicate the presence of self-modifying and indirect jump exploit code. The variable can infect others through data transfer instructions (`push`, `pop`, `move`) and instructions that perform arithmetic or bit-logic operations (`add`, `sub`, `xor`).

The method uses initialization analysis in order to reduce the false positive rates. The analysis is based on the assumption that the operands of self-modifying code and code using the indirect transitions, must be initialized. If not, object s is considered as legitimate. Formally, $P_1 = tainted(S)$, $P_2 = initialized(S)$,

$$|\tilde{\alpha}^{STILL}(S)| \neq 0 \Leftrightarrow P_1(S) \& \neg P_2(S).$$

Semantic-aware malware detection [19] is a signature-based approach. The method creates a set of behavior signature patterns by training on a sample of malicious objects. The object S is considered as malware if its behavior conforms to at least one pattern from this set.

In [19] authors have proved the following theorem:

Theorem 4: The problem of determining whether S satisfies a template T of a program behavior is undecidable.

Thus, the authors notice that their method can not have full coverage of classes of malicious programs. The method identifies a malicious object to a limited number of program modification techniques. The algorithm constructs a set $\{T\}$ of patterns of a programs malicious behavior. It is believed that the object S matches the pattern, if the following conditions are satisfied:

- The values in the addresses, which were modified during execution, are the same after the template execution with the appropriate context;
- A sequence of system calls in template is a subsequence of system calls in S ;
- If the program counter at the end of executing the template T points to the memory area whose value changed, then the program counter after executing S should also point into the memory area whose value changed.

In order to check whether object S matches the behavior pattern, the method checks that the vertices of the template correspond to vertices of S . The method also implements the construction of "def-use" ways and its checking. *Matching of template nodes to program nodes* is carried out by constructing a control flow graph CFG, with respect to the following rules (we also describe the predicate $P_1(S)$ checking whether nodes match each other) :

- A variable in the template can be unified with any program expression, except for assignment expressions;
- A symbolic constant in the template can only be unified with constant in S ;
- The function memory can be unified with the function memory only;
- An external function call in the template can only be unified with the same external function call in the program.

Preservation of def-use paths. A def-use path is a sequence of template nodes (or $CFG(S)$). The first node of def-use path defines the variable and the last uses it. Each def-use path in a template should correspond to the program def-use path. Next, method checks whether a variable is stored in an invariant meaning or not in the paths. To solve the problem of preservation of the variable using the following procedures are implemented:

- first, the NOP-sled lookup using simple signature matching;
- second, search of such code fragments in which values of variables are not preserved. If found, the corresponding fragment of code is executed with a random initial state;
- finally, using the theorem prover like the Simplify method [20] or the UCLID method [21].

We define the predicate P_2 which checks the Preservation of def-use paths. Thus, $T \sim S \Leftrightarrow P_1(S) \& P_2(S)$. The algorithm itself can be formally described as following:

$$|\tilde{\alpha}^{Semantic_aware}(S)| \neq 0 \Leftrightarrow \exists T_i \in \{T\} : T_i \sim S.$$

D. Dynamic methods

One example of the dynamic method is the emulation method (**Emulation**) proposed by Markatos et al in [23]. The main idea of the approach is to analyze the chain of instructions received during execution in a virtual environment. The execution starts from each and every position of the input buffer since the position of the shellcode is not known in advance. Thus, the method generates a set of objects

$$\{S'_i \mid S'_i \subset S\}$$

from object S . If at least one of the objects S_i satisfies the following heuristics, object S is considered as malware. These heuristics include the execution of some form of getPC code by an execution chain of S'_i ; another heuristic is checking whether the number of the memory accesses excess a given threshold. The object S'_i is considered as legitimate if during its execution an incorrect or privileged instruction was met. Let us define the following predicates:

$$P_1(S_i) = getPC \in S_i \& mem_access_number(S_i) \geq Thr,$$

where Thr is threshold;

$$P_2(S_i) = \forall j : I_j \in S_i \& invalid(I_j).$$

Thus, [23] can be formally described as:

$$|\tilde{\alpha}^{Emulation}(S)| \neq 0 \Leftrightarrow \exists i : S_i \subset S \& P_1(S_i) \& \neg P_2(S_i).$$

Method **NSC emulation** [26] is an extension of [23]. The method focuses on non-self-contained (NSC) shellcode detection. The execution of executable chains also starts from each and every position of the input buffer. Object S is considered as malware, if it satisfies the following heuristic. Let *unique writes* be the write operations to different memory locations and let *wx-instruction* be an instruction that corresponds to code at any memory address that has been written during the chain execution. Let W and X be thresholds for the unique writes and *wx*-instructions, respectively. The object belongs to the class K_{NSC} , if after its execution emulator has performed at least W unique writes ($P_1(S) = unique_writes \geq W$) and has executed at least X *wx*-instructions ($P_2 = wx \geq X$). Thus,

$$|\tilde{\alpha}^{NSC}(S)| \neq 0 \Leftrightarrow P_1(S) \& P_2(S).$$

Another method, which uses emulation is **IGPSA** [25]. The information about instruction is processed by automaton. All the instructions are categorized into five categories, represented by patterns P_1, \dots, P_5 . If an instruction writes PC into certain memory location, it is categorized into P_1 ; if it reads PC from the memory, it belongs to P_2 ; if it reads from memory location the instruction sequence resides in, it belongs to P_3 ; if it writes data into memory location PC, it belongs to P_4 ; otherwise it belongs to P_5 . Method generates a sequence of transformed patterns W which consists of elements of the set $\{P_1, \dots, P_5\}$. Thus, the object classification problem is the problem of determining whether its transformed pattern sequence W is accepted by automaton

$$IGPSA = (Q, \Sigma, \delta, q_0, F),$$

where Q is the set of states, $\Sigma = \{P_1, \dots, P_5\}$ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, q_0 is the initial state and F is set of final states. Each state corresponds to polymorphic shellcode behavior. Let us describe the predicate $P(S)$ which checks whether W is accepted by IGPSA. Formally,

$$|\tilde{\alpha}^{IGPSA}(S)| \neq 0 \Leftrightarrow P(S).$$

E. Hybrid methods

One of the examples of the hybrid method is the method for detecting self-decrypting shellcode [24], proposed by K. Zhang. Let us call it **HDD** in the rest of the paper. The static part of the method includes two-way traversal and backward data flow analysis. By which the analysis method finds seeding subsets of instructions of S . The presence of malicious behavior is verified by the emulation of these subsets.

Firstly, static analysis method performs recursive traversal analysis of the instruction flow, starting at the seeding instruction. A seeding instruction that can demonstrate the behavior of *GetPC* code (for example, *call*, *fnstenv*, etc.). The method starts the backward analysis, if a target instruction, an instruction that is either (a) an instruction that writes to memory, or (b) a branching instruction with indirect addressing, is encountered during the forward traversal. The method follows backwards the def-use chain in order to determine the operands of the target instruction. Then the method checks such chains $S_i \subset \{two_way_analysis(S)\}$ for the presence of cycles ($P_1(S_i)$). Moreover, it checks whether chains write to memory in the code address space (that fact is considered as self-modification behaviour). Let it be the $P_2(S_i)$ predicate. Let us also consider

$$P_3(S_i) = \forall j : I_j \in S_i \ \& \ invalid(I_j).$$

Thus,

$$|\tilde{\alpha}^{Hybrid_dec_detection}(S)| \neq 0 \Leftrightarrow \exists i : S_i \subset \{two_way_analysis(S)\} \ \& \ P_1(S_i) \ \& \ P_2(S_i) \ \& \ \neg P_3(S_i).$$

Another hybrid method is **PolyUnpack** [27]. This method is based on stational constructing of a program model and verification of this model by the emulation technique. The object S is said to be legitimate if it does not produce any data to be executed. Otherwise, the object is a self-extracting program. At the stage of static analysis, the object S is divided into code blocks and data blocks. These code blocks, separated by blocks of data, are a sequence of instructions Sec_0, \dots, Sec_n , which represent the program's model. The statically derived model and object S are then transited into the dynamic analysis component where S is executed in an isolated environment. The execution is paused after each instruction and its execution context is compared with the static code model. If the instruction corresponds to the static model, then the execution continues. Otherwise, the object S is considered as malware. Let us describe the predicate $P(S)$ which checks whether object S satisfies its static code model. Then we can formally describe PolyUnpack as

$$|\tilde{\alpha}^{PolyUnpack}(S)| \neq 0 \Leftrightarrow \neg P(S).$$

F. Methods of abstract execution

At the present day, this class represented the only method that is called **APE** [28]. APE is NOP-sled detection method, which is based on finding sufficiently long sequences of valid instructions, whose operands in memory are in the protected address space of the process. There are a small number of positions in the experimental data, from which abstract execution should be started, which are chosen in order to reduce the computational complexity. The abstract execution is used to check the instruction's correctness and validity.

Definition 8: A sequence of bytes is correct, if it represents a single valid processor instruction. A sequence of

bytes is valid if it is correct and all memory operands of the instruction reference the memory addresses that the process which executes the operation is allowed to access.

The number of correct instructions, which are decoded from each selected position, are denoted as *MEL* (Maximum Executable Length). It is possible that a byte sequence contains several disjoint abstract execution flows and the *MEL* denotes the length of the longest. The NOP-sled is believed to be found in S , if the value of *MEL* reaches a certain threshold *Thr*. Formally,

$$|\tilde{\alpha}^{APE}(S)| \neq 0 \Leftrightarrow (MEL \geq Thr).$$

III. EVALUATION AND DISCUSSION

This section provides an analysis and comparison of the above methods with respect to three criteria: the completeness of classes handling, the false positive rates and the computational complexity. The key difficulty here is that all the research papers observed in this paper use completely different testing conditions and testing datasets. Therefore it is not very helpful to compare the published false positives rates or throughput of the algorithms. For example, the STRIDE method was tested using only HTTP URI dataset, where the possibility of finding executable byte sequences is indeed relatively low. Some of the methods like SigFree are designed to detect "meaningful" executables and distinguish them from random byte sequences which only look like executable, but such method would definitely have high false positive rates when used for shellcode detection in the network channel where ELF executable transfer is quite normal. This means that the results provided in the original research papers can not be used directly for solving the problem of aggregate classifier generation. A real performance and false positive profiling should be performed, with some kind of representative dataset and a solid experiment methodology.

But for the task of the survey and making some preliminary relative comparison of the detection methods within the same shellcode feature classes the data provided in the original research papers could be useful. Therefore, we collected it into a series of summary tables, along with short descriptions of the testing conditions. The computational complexity estimation was made using the algorithm descriptions from the research papers and general knowledge of computational complexity of the typical tasks like emulation or sandboxing. The drawback of such estimation is that it gives only classes of complexity, not the real throughput in any given conditions. The actual throughput of the considered methods was analytically evaluated for the normalized machine 2.53 GHz Pentium 4 processor and 1 GB RAM with running Linux on it. Throughput is considered below when discussing the advantages and disadvantages of the methods.

Table I shows the comparison results for the completeness of classes coverage.

	Buttercup	Hamsa	Polygraph	Stride	Racewalk	Styx	Structural analysis	SigFree	STILL	Semantic aware	Emulation	HDD	NSC	IGPSA	PolyUnpack	APE
K_{NOP_1}																
K_{NOP_2}																
K_{NOP_3}																
K_{NOP_4}																
K_{NOP_5}																
K_{NOP_6}																
K_{NOP_7}																
K_{SH}																
K_{DATA}																
K_{ALT_OP}																
K_R																
K_{ALT_I}																
K_{INJ}																
K_{SELF_UNP}																
K_{SELF_CIPH}																
K_{RET}																
K_{RET+}																
K_{MET}																
K_{NSC}																

Table I: Methods coverage evaluation

Table II shows the comparison results of false positive and false negative rates for the above methods. The rate was calculated for those classes of malicious objects, which were covered by the appropriate method. It is important to note the following fact. As table II shows, rates of false positive are low enough. Nevertheless, the number of false positives on the real channels reach very high values, because of the large volume of transmitted data.

Table III shows computational complexity of the methods.

We consider the methods in terms of their applicability to the analysis of traffic on high-speed channels, as well as provide deeper understanding the space of the algorithms, comparison and tradeoffs between them.

For example, it is known that the method **ButterCup** could detect the exploits with many kinds of obfuscation (see table I). But the method usage on real channels is problematic. This is due to the fact that the method uses signatures of the return address, but a static return address in the modern exploits isn't used. In addition, the ButterCup usage as the only one detection method implies a large number of false positives. Nevertheless, the method can be used as an additional check with other tools, as it doesn't require much time and computing costs. The method can be applied to channels with any traffic profile (with any probability of executable code will appear in the channel), as well as permits analysis of high-speed data in real time. Average throughput of the method, calculated analytically, is 4,34Mb/s.

Both **Polygraph** and **Hamsa** have similar pre-processing requirements. Both of these methods are based on the

automatic generation of context-dependent signatures and provide similar shellcode classes coverage. Nevertheless, the method Hamsa isn't suited for polymorphic versions of the virus detection because of specifics of generated signatures. Different kinds of Polygraph's signatures provide a more flexible method. Although, the polymorphic version of the virus isn't detected by Polygraph in general case. All three Polygraph's signature classes have advantages and disadvantages. The token-subsequence signatures are more specific than the equivalent conjunction signatures. However, some exploits may contain invariants that can appear in any order. In that case, the token-subsequence signatures are more preferable. The Bayes signatures are generated more quickly than the others and are more useful when the invariants arise in exploits some of the time. The authors recommend to use all three types of signatures at the same time, but it implies a large overhead. For example, Polygraph in the best case 64 times slower than the Hamsa algorithm, in the worst case this value reaches 361 times. Average throughput of the Hamsa, estimated analytically, is 7,35Mb/s which makes the method applicable in real-time analysis of high-speed traffic. Average throughput of Polygraph without clustering reaches the value of 10Mb/s, but the accuracy of the method decreases in the same time. Average throughput of the method with clustering reaches the value of 0.04Mb/s only. In that case method can be used only as off-line analyzer.

In contrast to the Hamsa and Polygraph, the **Structural analysis** method can detect some types of obfuscated shellcode. Moreover, in some cases it is able to detect

Method	FP, %	FN, %	Testing sets
Buttercup	0.01	0	TCPdump files of network traffic from the MIT Lincoln Laboratory IDS evaluation Data Set
Hamsa	0.7	0	Suspicious pool: Polygraphs pseudo polymorphic worms; polymorphic version of Code-Red II; polymorphic worms, created with CLET and TAPION; Normal traffic: HTTP URI
Polygraph	0.2	0	Malicious pool: the Apache-Knacker exploit, the ATPhttpd exploit, BIND-TSIG exploit; Network traces: 10-day HTTP trace (125,301 flows); 24-hour DNS trace
Stride	0.0027*	0	Malicious pool: sleds, generated by the Metasploit Framework v2.2; Network traffic: HTTP URI;
Racewalk	0.0058	0	Malicious pool: sleds, generated by the Metasploit Framework v2.2; Normal traffic: HTTP URI, ELF executables, ASCII text, multimedia, pseudo-random encrypted data.
Styx	0	0	Malicious pool: exploits generated using the Metasploit framework; Normal data: network traffic collected at a enterprise network, which is comprised mainly of Windows hosts and a few Linux boxes.
Structural	0.5	0	Malicious pool: malicious code that was disguised by ADMmutate; Normal traffic: data consists to a large extent of HTTP (about 45%) and SMTP (about 35%) traffic The rest is made up of a wide variety of application traffic: SSH, IMAP, DNS, NTP, FTP, and SMB traffic.
SigFree	0**	0	Malicious pool: unencrypted attack requests generated by Metasploit framework, worm Slammer, CodeRed Normal data: HTTP replies (encrypted data, audio, jpeg, png, gif and flash).
STILL	0**	0	Malicious pool: code that was generated using Metasploit framework, CLET, ADMmutate
Semantic aware	0	0	Malicious pool: set of obfuscated variants of B[e]agle; Normal data: set of 2,000 benign Windows programs
Emulation	0.004	0	Malicious pool: code generated by Clet, ADMmutate, TAPION and Metasploit framework; Normal data: random binary content
HDD	0.0126	0	Malicious pool: code generated by Metasploit Framework, ADMmutate and Clet; Normal data: UDP, FTP, HTTP, SSL, and other TCP data packets; Windows binary executables
NSC	0	0	Malicious pool: code generated by Avoid UTF8/tolower, Encoder and Alpha2 Normal data: three different kinds of random content such as binary data, ASCII-only data, and printable-only characters
IGPSA	0	0	Malicious pool: code generated by Clet, ADMmutate, Jempiscodes, TAPION, Metasploit Framework Normal data: two types of traffic traces: one contains common network applications HTTP and HTTPs, of port 80 and 443; the other contains traces of port 135, 139 and 445
PolyUnpack	0	0	Malicious pool: 3,467 samples from the OARC malware suspect repository.
APE	0	0	Malicious pool: IIS 4 hack 307, JIM IIS Server Side Include overflow, wu-ftp/2.6-id1387, ISC BIND 8.1, BID 1887 exploits; Normal data: HTTP and DNS requests.

Table II: Accuracy of the methods. FP stands for “False Positives” and FN stands for “False Negatives”

Method	Complexity	Remarks
Buttercup	$O(N)$	N is the length of S
Hamsa	$O(T \times N)$	N is the length of S , T is the number of tokens in signature
Polygraph	$O(N)$ $O(N + S^2)$ $O(M^2 \times L)$	without clusters N is the length S with clusters S is the number of clusters method’s training M the length of malware training information L - the length of legitimate training information
Stride	$O(N \times l^2)$	N is the length of S , l is the length of NOP-sled
Racewalk	$O(N \times l)$	N is the length of S , l is the length of NOP-sled
Styx	$O(N)$	N is the length of S
Structural	$O(N)$	N is the length of S
SigFree	$O(N)$	N is the length of S
STILL	$O(N)$	N is the length of S
Semantic	$O(N)$	N is the length of S
Emulation	$O(N^2)$	N is the length of S
HDD	$O(N + K^2 \times T^2)$	N is the length of S K is the number of suspicious chains T is maximum length of suspicious chains
NSC	$O(N^2)$	N is the length of S
IGPSA	$O(N^2)$ $O(CN)$	non-optimized optimized
PolyUnpack	$O(N)$	N is the length of S
APE	$O(N \times 2^l)$	N is the length of S l is the length of NOP-sled

Table III: Methods complexity

metamorphic shellcode as it generates program's structure dependent signatures. In spite of the fact the algorithmic complexity of all three algorithms is comparable (see table III), Structural analysis slower than the others. Because of the time complexity of algorithm, traffic analysis is possible in off-line mode only. Average throughput of the method reaches the value of $1Mb/s$. In addition, technique cannot detect malicious code that consists of less than k blocks. That is, if the executable has a very small footprint method cannot extract sufficient structural information to generate a fingerprint. The authors chose 10 for k in their experiments.

The **Racewalk** method improves the **Stride** algorithm by significantly reducing of computational complexity. Both Racewalk and Stride can be used in real-time analysis of high-speed channels. When comparing the methods of false positives rate it is necessary to consider the following observation for the Stride algorithm. There is a possibility that NOP-equivalent byte sequence can occur in legitimate traffic. For example, a sequence of bytes may appear as part of ELF executable, ASCII text, multimedia or pseudo-random encrypted data. Thus, the value presented in Table II for this type of legitimate traffic may vary from what is represented. Both of these methods significantly exceed the speed of the **APE** method of abstract interpretation which also detects NOP-sled. In that case it is difficult to use APE on real channels.

The **Styx** method is able to detect self-unpacked and self-ciphered shellcode. Nevertheless, in the average case Styx is slower than similar methods of dynamic analysis. Particularly, the average throughput of the method is $0.002Mb/s$. That significantly decreases the method's applicability. Nevertheless, it can be used as a supplement to other shellcode detection algorithms. The method as an additional tool to others can increase the shellcode space coverage. Another considered method which is based on CFG construction is **Semantic aware algorithm**. It is also characterized by low-speed analysis. In that case the method cannot be used in real-time mode even on channels with low bandwidth. The second limitation of method comes from the use of def-use chains. The def-use relations in the malicious template effectively encode a specific ordering of memory updates. Thus, the algorithm can detect only those program that exhibit the same ordering of memory updates. Nevertheless, the method can be used as additional checking tool to others shellcode detection algorithms.

Methods **SigFree** and **STILL** together providing particularly complete coverage of all shellcode classes. In addition, methods are able to work in real-time mode on high-speed channels. However, the value of false positives rates of SigFree and STILL methods represent only the traffic profile, which does not allow any kind of executables. For the other traffic profile false positive rates of these methods are extremely high. That fact decreases the applicability of SigFree and STILL.

Significant advantage of methods **Emulation**, **NSC Emulation**, **IGPSA** is their resistance to anti-static evasion techniques. At the same time, all these methods have a limited applicability since they can detect only shellcode classes that contain anti-static obfuscation. As example, the Emulation method detects only polymorphic shellcodes that decrypt their body before executing their actual payload. Plain or completely metamorphic shellcodes that do not perform any self-modifications are not captured by algorithm. However, polymorphic engines are becoming more prevalent and complex. The method's throughput is analytically evaluated as $1Mb/s$. Method NSC Emulation, running at average throughput $1.25 - 1.5Mb/s$ is focused on finding non-self-contained shellcode which practically doesn't occur in real traffic. Thus, the applicability of the method isn't clear. The average throughput of IGPSA algorithm is $1.5Mb/s$. Algorithms IGPSA and Emulation can interchanged with each other.

Average estimated throughput of the hybrid method **HDD** is $1.5Mb/s$. That allows to use the method on the channels characterized by a relatively low bandwidth in real-time mode. An important advantage of the method is its ability to detect metamorphic shellcode, along with other classes that use anti-static obfuscation techniques. However, the authors didn't test the method on non-exploit code that uses code obfuscation, code encryption, and self-modification. That fact can potentially change the false positives rate proposed by the authors. Thus, this is true for the other methods which detects polymorphic and metamorphic shellcodes.

The throughput of **PolyUnpack** hybrid method is significantly lower than HDD and estimated as $0.05Mb/s$. This is due to time requirement to model generation and long delays between running program request and model response. In addition, with decreasing of the program size, the throughput of method decreases respectively. Nevertheless, the method characterized by 100% detection accuracy and zero false positives rate. That makes possible to use method as an additional analyzer to other shellcode detection algorithms.

IV. PROPOSED APPROACH AND CONCLUSION

This paper discusses techniques to detect malicious executable code in high-speed data transmission channels. Malicious executable code is characterized by a certain set of features by which the entire set of malware can be divided into the classes. Thus, the problem of shellcode detection can be formulated in terms of recognition theory. Each shellcode detection method can be considered as a classifier which assigns the executable malicious code to one of the classes K_i of shellcode space. Each classifier has its own characteristics of shellcode space coverage, false negative and false positive rates, computational complexity.

Using the set of classifiers we can formulate the problem of automatic synthesis of such hybrid shellcode detector,

which will cover all shellcode feature classes and reduce the false positive rates while reducing the computational complexity of the method compared with the simple linear combination of algorithms. The method should be synthesized in conformance with the profile of traffic channel data. In other words, the method should consider the probability of executable code transmission through the channel, etc. Let us consider the problem of algorithm synthesis as construction of a directed graph $G = (V, E)$ (see Fig. 3) with a specific topology, where $\{V\}$ is the set of nodes which are classifiers themselves, $\{E\}$ is the set of arcs. Each arc represents the route of flow data. We decided to include in the graph such classifiers (methods) that provide the most complete coverage of the shellcode classes K_1, \dots, K_l . Each of the selected classifiers is assigned with two attributes: false positive rates and complexity. The attributes' values can be calculated by profiling, for example.

This qualifier must change the corresponding bit in the information vector from the delta to 0 or 1. If the corresponding bit different from the delta, the classifier produces for him a logical or operation: $\alpha_r(S) = \alpha_{TCURRENT}(S) \parallel \alpha_{TPREVIOUS}(S)$.

Each arc (v_i, v_j) is marked with one of the classes K_r if v_i classifier checks whether the object (flow data) belongs to class K_r . The v_i classifier changes the corresponding bit $\alpha_r(S)$ in the information vector $\tilde{\alpha}(S) = (\alpha_1(S), \alpha_2(S), \dots, \alpha_l(S))$ from Δ to value from $\{0, 1\}$. If $\alpha_r(S) \neq \Delta$ then the classifier produces for it a logical OR operation: $\alpha_r(S) = \alpha_{TCURRENT}(S) \parallel \alpha_{TPREVIOUS}(S)$. If the classifier v_i checks whether the object S belongs to several classes of shellcode space, then the vertex v_i has several outgoing arcs with the corresponding notes. Similarly, the classifier changes the values of corresponding bits in information vector $\tilde{\alpha}(S)$. In addition, if vertex v_i has several incoming arcs, then the results of classifiers, from which the arcs are outgoing, merge with each other.

We assume that each node is associated with the type of the set $\{REDUCING, NON_REDUCING\}$. If a node v_i has type *REDUCING*, then if the classifier v_i concludes object S to be legitimate, the flow is not passed on. That implies the computational cost decreases and input flow is reduced. The reduced flow example is shown in Fig. 4

We associate each path in the graph G with its weight. The weight consists of a combination of two parameters: i) the total processing time, and ii) the false positive rates. it is necessary to include a classifier with lowest false positive rates to each path in G .

As part of the problem being solved it is necessary to propose a topology of graph G such that: i) the traffic profile will be taken into account; ii) all pathes will be completed in the shortest time, and iii) all pathes will be completed with the lowest false positive rates. We will consider that problem in terms of multicriteria optimization theory.

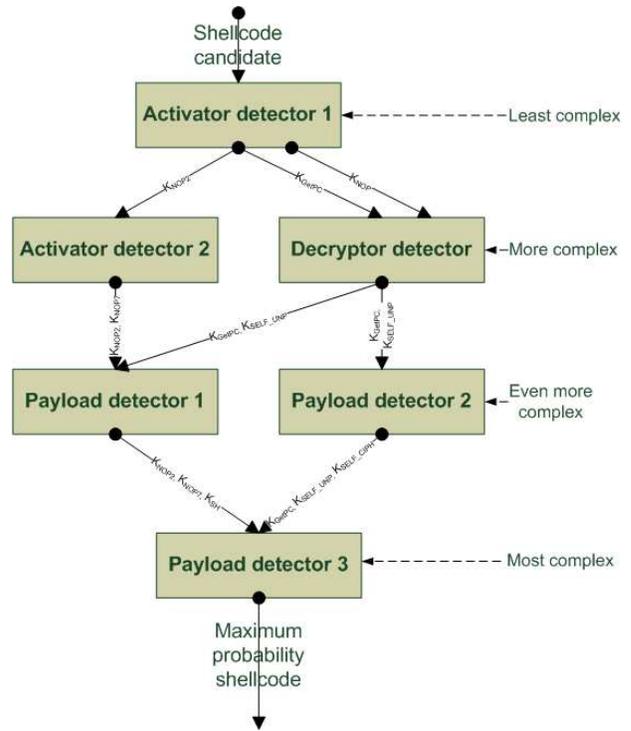


Figure 3: Graph example. Solid arrow represents the route of shellcode candidates. The arc (v_i, v_j) is marked with one of the classes K_x if v_i classifier checks whether shellcode candidate belongs to class K_x .

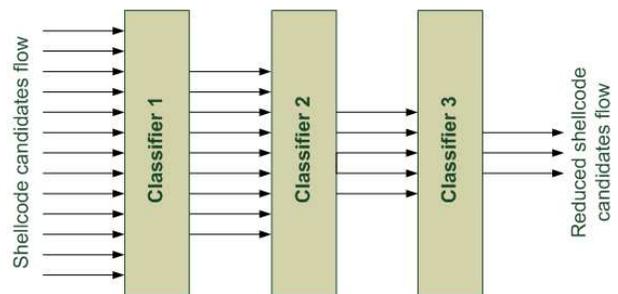


Figure 4: Example of flow reducing. Arrows represent the flow of shellcode candidates. The Classifiers 1, 2 and 3 consider part of the objects as legitimate, so they are not passed on.

REFERENCES

- [1] Y. I. Zhuravlev, *Algebraic approach to the solution of recognition or classification problems*. Pattern recognition and image analysis, 1998, vol. 8; no.1, 59-100.
- [2] Team Cymru Malware Infections Market. [PDF] <http://www.team-cymru.com/ReadingRoom/Whitepapers/2010/Malware-Infections-Market.pdf>

- [3] B. Stone-Gross et al., *Your Botnet is My Botnet: Analysis of a Botnet Takeover*. Technical report, University of California, May 2009.
- [4] K. Kruglov, *Monthly Malware Statistics: June 2010*. Kaspersky Lab Report, June 2010. [HTML] http://www.securelist.com/en/analysis/204792125/Monthly_Malware_Statistics_June_2010
- [5] P. Porras, H. Saidi, V. Yegneswaran, *An Analysis of Conficker's Logic and Rendezvous Points*. Technical Report, SRI International, Feb 2009.
- [6] FBI, *International Cooperation Disrupts Multi-Country Cyber Theft Ring*. Press Release, FBI National Press Office, Oct 2010.
- [7] U. Payer, M. Lamberger, P. Teufl, *Hybrid engine for polymorphic shellcode detection*. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA05). Berlin: Springer-Verlag, 2005. 19-31
- [8] R. Chinchani, E. Berg, *A fast static analysis approach to detect exploit code inside network flows*. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05). Berlin: Springer-Verlag, 2005. 284-308
- [9] C. Kruegel, E. Kirda, D. Mutz, et al., *Polymorphic worm detection using structural information of executables*. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05). Berlin: Springer-Verlag, 2005
- [10] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis, *Stride: Polymorphic sled detection through instruction sequence analysis*. In Proc. of the 20th IFIP International Information Security Conference (SEC'05), 2005.
- [11] D. Gamayunov, N. T. Minh Quan, F. Sakharov, E. Toroshchin *Racewalk: fast instruction frequency analysis and classification for shellcode detection in network flow* In: 2009 European Conference on Computer Network Defense. Milano, Italy, 2009
- [12] A. Pasupulati, J. Coit, K. Levitt, et al., *Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities*. In: Proceedings of Network Operations and Management Symposium 2004. Washington: IEEE Computer Society, 2004
- [13] E. Filiol, *Metamorphism, formal grammars and undecidable code mutation*. International Journal of Computer Science, 2, 2007
- [14] Z. Li, M. Sanghi, Y. Chen, et al., *Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience*. In: Proceedings of 2006 IEEE Symposium on Security and Privacy (S&P'06). Washington: IEEE Computer Society, 2006. 32-47
- [15] J. Newsome, B. Karp, D. Song, *Polygraph: automatically generating signatures for polymorphic worms*. In: Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05). Washington: IEEE Computer Society, 2005. 226-241
- [16] M. Weiser, *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979
- [17] X. Wang, C. C. Pan, P. Liu, S. Zhu, *Sigfree: A signature-free buffer overflow attack blocker*. In 15th Usenix Security Symposium, July 2006
- [18] X. Wang, Y. Jhi, S. Zhu, *Protecting Web Services from Remote Exploit Code: A Static Analysis Approach* In Proc. of the 17th international conference on World Wide Web (WWW'08), 2008.
- [19] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, *Semantics-aware malware detection*. In Proc. of 2005 IEEE Symposium on Security and Privacy (S&P'05), 2005.
- [20] D. Detlefs, G. Nelson, J. B. Saxe *Simplify: A Theorem Prover for Program Checking*
- [21] R. E. Bryant, S. k. Lahiri, S. A. Seshia, *Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions*. In: CAV 02: International Conference on Computer-Aided Verification
- [22] A. Stavrou, M. E. Locasto, Y. Song, *On the Infeasibility of Modeling Polymorphic Shellcode* In Proc. of the 14th ACM conference on Computer and communications security (CCS'07), 2007.
- [23] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, *Network-level polymorphic shellcode detection using emulation*. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Berlin: Springer-Verlag, 2006
- [24] Q. Zhang, D. S. Reeves, P. Ning, et al., *Analyzing network traffic to detect self-decrypting exploit code*. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, New York: ACM, 2007. 4-12
- [25] L. Wang, H. Duan, X. Li, *Dynamic emulation based modeling and detection of polymorphic shellcode at the network level* Science in China Series F: Information Sciences Volume 51, Number 11, 1883-1897.
- [26] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos *Emulation-based Detection of Non-self-contained Polymorphic Shellcode* In Proc. of the 10th international conference on Recent advances in intrusion detection (RAID'07), 2007.
- [27] P. Royal, M. Halpin, D. Dagon, R. Edmonds, W. Lee, *PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware* In: Computer Security Applications Conference (ACSAC'06), 2006.
- [28] T. Toth, C. Kruegel, *Accurate Buffer Overflow Detection via Abstract Payload Execution* In Proc. of the 5th international conference on Recent advances in intrusion detection (RAID'02), 2002.

Station Disassociation Problem in Hosted Network

Artyom Shal

Software Engineering department
The Higher School of Economics
Moscow, Russia
artiom.shal@gmail.com

Abstract — hosted network technology gives an opportunity to create the virtual access point. However, client stations are forced to disassociate from AP due to poor configuration. This paper proposes solution to the issue of station disassociation in the hosted networks.

Keywords — hosted network, Wi-Fi, TCP/IP.

I. INTRODUCTION

Microsoft hosted network is not a new technology, however, it is not examined enough yet. Although it gives the opportunity to arrange fully qualified access point with no additional hardware required, users face connectivity problems too often. Virtual access point drops the connection with users' PC frequently for no apparent reason (at first sight). This action is very disturbing and disruptive. This paper is aimed to uncover possible issues and pitfalls of the powerful technology.

II. FIRST INVESTIGATION

The first apparent reason for station disassociating from virtual AP is that TCP and Wi-Fi are not perfectly combined. The nature of the IEEE 802.11 technology causes packet delay and loss rate, which triggers TCP congestion control mechanism [1]. This may lead to performance degradation. However, connection breaks were not reported on such scenarios. Possible reason for connection breaks could be specific features of Microsoft TCP/IP stack. Virtual access point may behave differently compared to real devices, indeed. Responsibility for smooth operation of the hosted network technology is on NIC manufacturers. Hence, we assume that virtual AP is fully compliant to 802.11 set of standards. What is the reason for such behavior?

A. Testing

A clear behavior pattern was discovered after performing tests. The testing involved *Microsoft Network Monitor* tool for capturing and analyzing wireless network traffic. The monitoring showed that the connection was fine when the heavy data transfer existed, e.g. video stream. On the other hand, when there was no network activity for more than 10 seconds connection was breaking. This is a rather infrequent situation for ordinary users, as many network services (like NetBIOS) on client stations usually communicate with each other. Yet, it is common enough for corporate environment, where security policies prohibit using many services. This

may lead to total blackout in network activity and thus to frequent connection breaks.

To measure the issue we used Windows API on the side of virtual AP. To monitor Wi-Fi network events, we registered system notifications from miniport driver. For this, we used *WlanRegisterNotification* function. The function was called in the following way:

```
DWORD prevNotif = 0;
DWORD lastError = WlanRegisterNotification(
    handle(),
    WLAN_NOTIFICATION_SOURCE_ALL,
    TRUE, //Ignore duplicate
    (WLAN_NOTIFICATION_CALLBACK)handleNotification,
    NULL,
    NULL,
    &prevNotif
);
```

To get the state of Wi-Fi NIC we handled specific message type in *handleNotification* callback function.

```
VOID handleNotification(WLAN_NOTIFICATION_DATA
*wlanNotifData, VOID *p)
{
    switch(wlanNotifData->NotificationSource) {
        case WLAN_NOTIFICATION_SOURCE_HNWK:
            switch(wlanNotifData->NotificationCode){
                case wlan_hosted_network_peer_state_change:
                    ...
            }
        ...
    }
}
```

Three devices were used for testing:

- Dell Latitude E5420 laptop with NIC Intel Centrino Advanced-N 6205
- Samsung Galaxy S3 smartphone with Samsung Exynos 4 Quad system on chip
- Macbook Air 13 laptop with NIC Realtek RTL8188CU Wireless LAN 802.11n

The results for Samsung device were the worst. It couldn't connect to access point at all. Dell device with Windows 7 OS on board showed good results. Virtually no disassociations were detected. Macbook Air laptop was attempting to reconnect the access point every 10-15 seconds (Fig.1).

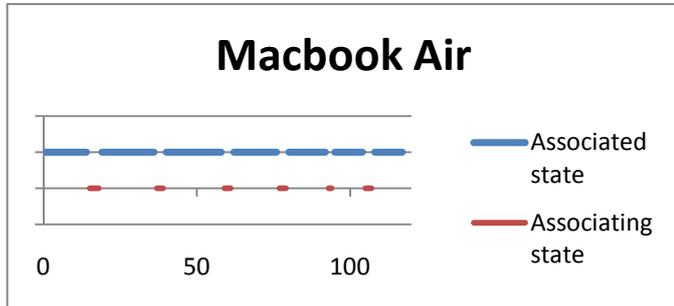


Fig 1. Macbook Air connection state pattern

B. DHCP server integration

The easiest way to fix this issue is to send stub packets. One candidate is ICMP packets used by ping utility. However, the obstacle is that we need to know the IP address of every connected client. The only way to know this at application layer is to allocate IP addresses dynamically through DHCP server.

Starting the wireless Hosted Network typically involves the launch of Internet Connection Sharing (ICS) service in standalone mode. This, in turn, leads to DHCPv4 server to begin providing private IPv4 addresses to connected devices. In this mode, only the DHCPv4 server is operating. This is a special operation mode for ICS and is only made available through the wireless Hosted Network. A user or application are not able to directly start and stop standalone ICS through public ICS APIs or *netsh* commands. Moreover, there are no ways to manage DHCP server operation. Therefore we had to stop ICS manually and to use some open source alternative.

To stop ICS in standalone mode we used simple workaround: the connected key in Windows registry was deleted. The OpenDHCP server is a good alternative to ICS DHCP server. After small modifications, we obtained the following result. The server was receiving the acknowledgement message and starting the ping utility. This simulated activity was enough to keep client stations connected.

C. Results

The tests of the modified DHCP server showed a much more steady connection for many devices. Still, the results were disappointing, as connection was still breaking. Deeper testing with wider variety of devices and different usage scenarios revealed that problem is not at transport or network layer [2] of the OSI model. It is somewhere at the underlying layers.

III. DEEPER EXPLORATION

To uncover the issue of station disassociation at data link layer we had to use special hardware and software. In particular, Proxim ORINOCO wireless network interface card was used to capture WLAN frames. This NIC can work in promiscuous mode, which makes the controller pass all received traffic to the central processing unit (instead of only passing the frames that the controller intended to receive). To monitor network activity the CommView software was used. The application has WLAN-specific features, such as displaying and decoding of management and control frames.

A. Monitoring

Tests showed that the disassociation frame (sent from virtual access point) was the reason for dropping connection. The reason field in that frame was "disassociated due to inactivity". This indicates that either the station's NIC is not sending probe frames frequent enough or that software-based AP cannot see them. We think that the latter is more likely due to specific feature of SoftAP—it shares the common processing unit with virtual station adapter. If proper buffering on NIC is not present, this may lead to a situation when AP is halted to process station operations and cannot process its own frames. This might be fixed by proper configuration during the process of association and initial handshake [3]. The virtual access point should notify the stations that more frequent probe requests are needed, as the listen intervals decreased.

B. Solution

As Microsoft doesn't provide any public API to configure hosted network we can manage it only through NDIS driver stack. NDIS stack has several types of drivers: protocol, miniport and filter (intermediate). Miniport driver is the prerogative of NIC manufacturer, so the filter driver is an appropriate tool to interact and affect the adapter.

The miniport driver notifies the filter driver on every event including virtual AP events. Using lightweight filter driver we modified the listen interval in beacon frames. This frames set short packet buffer, which forced the client stations to make probe requests more frequently. This prevents the AP from sending disassociation requests.

To access the configuration of the beacon frames we used the `OID_DOT11_BEACON_PERIOD` object type, which requests the miniport driver to set specified value of the IEEE 802.11 at *dot11BeaconPeriod management information base (MIB) object*. This object is used by the 802.11 station for scheduling the transmission of 802.11 beacon frames. It also represents the *Beacon Interval* field of the 802.11 Beacon and Probe Response frames sent by the station.

The data type for `OID_DOT11_BEACON_PERIOD` is a ULONG value that specifies the beacon period in 802.11 time units (TU). One TU is 1024 microseconds. The `dot11BeaconPeriod MIB object` has a value from 1 through 65535.

C. Results

The tests showed a stable connection for all reference devices. devices operating in power saving mode (Samsung Galaxy S3) can go asleep in ATIM window if there are no announcements [4]. However, right configured beacon intervals fixed this issue.

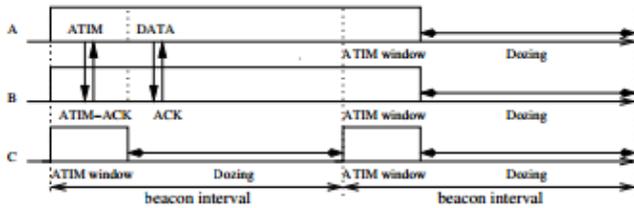


Fig 2. ATIM messages announcement

Another specific case is channel switch. The device that has the ability to scan networks in background may switch channels for short periods. Devices with this feature (Macbook Air) may miss beacon announcement. NDIS configuration fixes this issue as well.

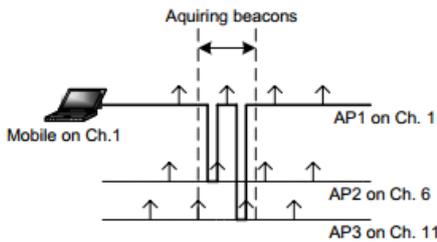


Fig 3. Channel switch for network scan

IV. CONCLUSION

It is clear that hosted network have some pitfalls. It is advisable for NIC manufacturers to provide not only standard-compliant devices, but also drivers that can avoid many pitfalls of the 802.11 protocol stack. The solution for station disassociation issue was given in this paper. It may find application in hot spot software.

REFERENCES

- [1] M. Franceschinis, M. Mellia, M. Meo, M. Munafo Measuring "TCP over WiFi: A Real Case," 1st workshop on Wireless Network Measurements (Winmee), Riva Del Garda
- [2] V. P. Kemerlis , E. C. Stefanis , G. Xylomenos , G. C. Polyzos "Throughput Unfairness in TCP over WiFi" Proc. 3rd Annual Conference on Wireless On demand Network Systems and Services (WONS 2006)
- [3] V. Gupta, M. K. Rohil, "Information Embedding in IEEE 802.11 Beacon Frame," Proc. National Conference on Communication Technologies & its impact on Next Generation Computing CTNGC 2012
- [4] H. Coskun, I. Schieferdecker, Y. Al-Hazmi "Virtual WLAN: Going beyond Virtual Access Points" Electronic Communications of the EASST, Volume 17, 2009
- [5] P. Bahl "Enhancing the Windows Network Device Interface Specification for Wireless Networking", Microsoft Research

On Bringing Software Engineering to Computer Networks with Software Defined Networking

Alexander Shalimov

Applied Research Center for Computer Networks,
Moscow State University
Email: ashalimov@arccn.ru

Ruslan Smeliansky

Applied Research Center for Computer Networks,
Moscow State University
Email: smel@arccn.ru

Abstract—The software defined networking paradigm becomes more and more important and frequently used in area of computer networks. It allows to run software that manages the whole network. This software becomes more complicated in order to provide new functionality that was impossible to imagine before. It requires better performance, better reliability and security, better resource utilization that will be possible only by using advanced software engineering techniques (distributed and high availability systems, synchronization, optimized Linux kernel, validation techniques, and etc).

I. INTRODUCTION

Software Defined Networking (SDN) is the "hottest" networking technology of recent years [1]. It brings a lot of new capabilities and allows to solve many hard problems of legacy networks. The approach proposed by the SDN paradigm is to move network's intelligence out from the packet switching devices and to put it into the logically centralized controller. The forwarding decisions are done first in the controller, and then moves down to the overseen switches which simply execute these decisions. This gives us a lot of benefits like global controlling and viewing whole network at a time that helpful for automating network operations, better server/network utilization, and etc.

A controller (also known as network operating system) is a dedicated host which runs special control software, framework, which interacts with switching devices and provides an interface for the user-written management applications to observe and control the entire network. In other words, the controller is the heart of SDN networks, and its characteristics determine the performance of the network itself.

We describe the basic architecture of contemporary controllers. For each part of a controller we show software engineering techniques are already used and might be used in the future in order to improve the performance characteristics.

We show the result of our latest experimental evaluation of SDN/Openflow controllers. Based on this we explain that the performance of single controller is not yet enough to manage data centers and large-scale networks.

Finally, we present the approach of high performance and reliable next generation distributed controller. We discuss possible ways to organized it and mention highly demands software engineering techniques.

II. BACKGROUND

A. History

Since early 2000th many researchers in Stanford University and Berkeley University have started rethinking the design and architecture of networking and Internet. The modern Internet and enterprise networks have a very complex architecture and are build using an old design paradigm. This paradigm includes the request for decentralized and autonomous control mechanisms which means that each network device implements both the forwarding functionality and the control plane (routing algorithms, congestion control, etc). Furthermore, any additional functionality in modern networking (for example, load balancing, traffic engineering, access control etc) is provided by the set of complex protocols and special gateway-like devices.

The enterprise and backbone networks, data center infrastructures, networks for educational and research organizations, home and public networks both wired and wireless are build upon a variety of proprietary hardware and software which are cost expensive and difficult to maintain and manage. This leads to inefficient physical infrastructure utilization, high oncost for management tasks, security risks and other problems.

Enterprise networks are often large, run a wide variety of applications and protocols, and typically operate under strict reliability and security constraints; thus, they represent a challenging environment for network management. The stakes are high, as business productivity can be severely hampered by network misconfigurations or break-ins. Yet the current solutions are weak, making enterprise network management both expensive and error-prone. Indeed, most networks today require substantial manual configuration by trained operators to achieve even moderate security [1], [3].

The Internet architecture is closed for innovations [4]. The reduction in real-world impact of any given network innovation is because the enormous installed base of equipment and protocols, and the reluctance to experiment with production traffic, which have created an exceedingly high barrier to entry for new ideas. Today, there is almost no practical way to experiment with new network protocols (e.g., new routing protocols, or alternatives to IP) in sufficiently realistic settings (e.g., at scale carrying real traffic) to gain the confidence needed for their widespread deployment. The result is that most new ideas from the networking research community go untried and untested.

Modern system design often employs virtualization to decouple the system service model from its physical realization. Two common examples are the virtualization of computing resources through the use of virtual machines and the virtualization of disks by presenting logical volumes as the storage interface. The insertion of these abstraction layers allows operators great flexibility to achieve operational goals divorced from the underlying physical infrastructure. Today, workloads can be instantiated dynamically, expanded at runtime, migrated between physical servers (or geographic locations), and suspended if needed. Both computation and data can be replicated in real time across multiple physical hosts for purposes of high-availability within a single site, or disaster recovery across multiple sites. Unfortunately, while computing and storage have fruitfully leveraged the virtualization paradigm, networking remains largely stuck in the physical world [6], [7], [8]. As is clearly articulated in [5], networking has become a significant operational bottleneck.

While the basic task of routing can be implemented on arbitrary topologies, the implementation of almost all other network services (e.g., policy routes, ACLs, QoS, isolation domains) relies on topology-dependent configuration state. Management of this configuration state is cumbersome and error prone adding or replacing equipment, changing the topology, moving physical locations, or handling hardware failures often requires significant manual reconfiguration.

Virtualization is not foreign to networks, as networking has long supported virtualized primitives such as virtual links (tunnels) and broadcast domains (VLANs). However, these primitives have not significantly changed the operational model of networking, and operators continue to configure multiple physical devices in order to achieve a limited degree of automation and virtualization. Thus, while computing and storage have both been greatly enhanced by the virtualization paradigm, networking has yet to break free from the physical infrastructure. Furthermore, the network virtualization functionality implemented via additional protocols under L2-L4 layers increase the complexity and cost of network hardware and the difficulty of configuring such hardware.

B. SDN

Further, to solve all above mentioned problems with network management and configuration, reduce the complexity of network hardware and software and make networks more open to innovations the broad community of academical and industrial researchers Open Networking Foundation [9] propose a new paradigm for networking the Software Defined Networking (SDN).

The approach proposed by the SDN paradigm is to separate the control plane (i.e. the policy for management network traffic) from the datapath plane (i.e. the mechanisms for real packet forwarding) (see Figure 1).

Traditionally, hardware implementations have embodied the logic required for packet forwarding. That is, the hardware had to capture all the complexity inherent in a packet forwarding decision. According to new paradigm [1], [2], [4] all forwarding decisions are done first in software (remote controller), and then the hardware merely mimics these decisions for subsequent packets to which that decision applies (e.g., all

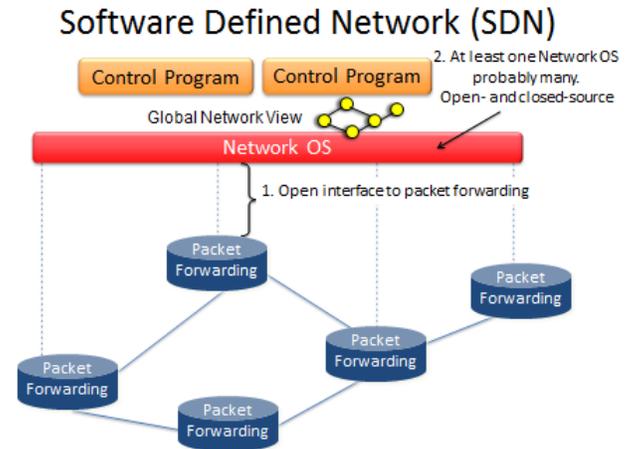


Fig. 1. Software Defined Network organization.

packets of given network flow). Thus, the hardware does not need to understand the logic of packet forwarding, it merely caches the results of previous forwarding decisions (taken by software) and applies them to packets with the same headers.

The key task is to match incoming packets to previous decisions. Packet forwarding is treated as a matching process, with all packets matching a previous decision handled by the hardware, and all non-matching packets handled by the software of remote controller. It is important to mention, that only packet headers are used in matching process.

A network switching hardware now must implement only a simple set of primitives to manipulate packet headers (match them against matching rules and modify if needed) and forward packets [1]. The core feature of such SDN-base switching software is a flow table which stores the matching rules (in form of packet header patterns to match against the incoming packet headers) and set of actions which must be applied to successfully matched packet.

Switching hardware also must provide common and vendor-agnostic interface for remote controller. To unify the interface between the switching hardware and remote controller the special OpenFlow protocol [10] was introduced. This protocol provides the controller a way to discover the OpenFlow-compatible switches, define the matching rules for the switching hardware and collect statistics from switching devices.

Figure 2 shows an interaction between OpenFlow-based controller and OpenFlow-based switching hardware, there controller provides the switch with a set of forwarding rules.

The control functionality in SDN paradigm is implemented by the remote controller a dedicated host which runs special control software. At the present time there exist a number of controllers. The most well known are NOX [12], POX [13], Beacon [14], Floodlight [15], MUL [16], Ryu [19], and Maestro [18]. Again, a controller is a framework which interacts with OpenFlow-compatible switching devices and provides an interface for the user-written management applications to observe and control the entire network. A controller does not

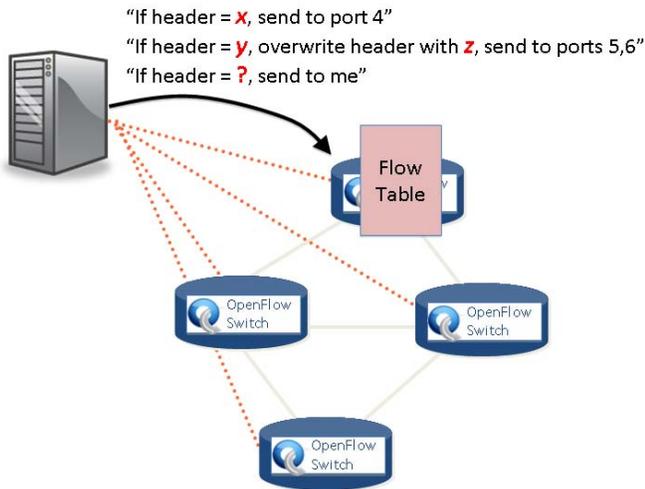


Fig. 2. Software Defined Network paradigm. Remote controller provides the forwarding hardware with rules describing how to forward packets according to their headers.

manage the network itself; it merely provides a programmatic interface. Applications implemented on top of the Network Operating System perform the actual management tasks.

A controller represents two major conceptual departures from the status quo. First, the Network Operating System presents programs with a centralized programming model; programs are written as if the entire network were present on a single machine (i.e., routing algorithms would use Dijkstra to compute shortest paths, not Bellman-Ford). Second, programs are written in terms of high-level abstractions (e.g., user and host names), not low-level configuration parameters (e.g., IP and MAC addresses). This allows management directives to be enforced independent of the underlying network topology, but it requires that the Network Operating System carefully maintain the bindings (i.e., mappings) between these abstractions and the low-level configurations.

C. OpenFlow

The OpenFlow protocol is used to manage the switching devices: adding new flow, deleting the flow, get statistics, and etc. It supports three message types: controller-to-switch, asynchronous, and symmetric.

Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. Asynchronous messages are initiated by the switch and used to update the controller of network events and changes to the switch state. Symmetric messages are initiated by either the switch or the controller and sent without solicitation.

The full set of messages and the detailed specification of OpenFlow protocol could be found in [11].

III. CONTROLLER

Based on analyzing available materials about almost twenty four SDN/OpenFlow controllers, we proposed the reference architecture of SDN/OpenFlow controller shown on Figure 3.

The main components are:

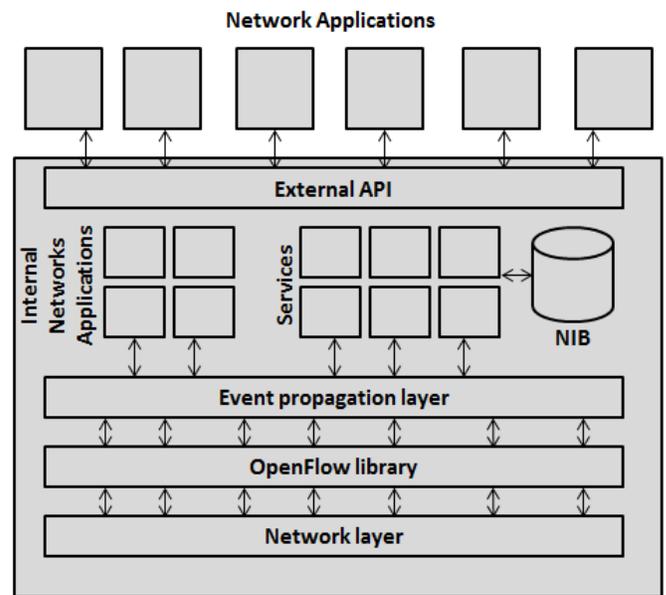


Fig. 3. The basic architecture of an OpenFlow/SDN controller.

- 1) Network layer is responsible for communication with switching devices. It is the core layer of every controller that determines its performance. There are two main tasks here:
 - Reading incoming OpenFlow messages from the channel. Usually this layer relies on the runtime of chosen programming language. For faster communication with NIC we can also use fast packets processing framework like netmap [20] and Intel DPDK [21].
 - Processing incoming OpenFlow messages. The common approach is to use multi-threading. One thread listens the socket for new switch connection requests and distributes the new connections over other working threads. A working thread communicates with the appropriate switches, receives flow setup requests from them and sends back the flow setup rule. There are a couple of advanced techniques. For instance, Maestro distributes incoming packets using round-robin algorithm, so this approach is expected to show better results with unbalanced load.
- 2) OpenFlow library. The main functionality is parsing OpenFlow messages, checking the correctness, and according to a packet type producing new event like "packetin", "portstatus", and etc. The most interesting part here, that is not in modern controllers yet, is resilience to incorrectly formed messages.
- 3) Event layer. The layer is responsible for event propagation between the controller's core, services, and network internal network applications. The network application subscribes on events from the core, produces other events to which other applications may subscribe. This is usually done by publishing/subscribing mechanism, either by writing your own implementation or using the standard one like

- libevent for C/C++, RabbitMQ for Erlang.
- 4) Services. This is the most frequently used network functionality like switches discovery, topology creating, routing, firewall.
- 5) Internal network applications. This is your-own application like L2 learning switch. "Internal" means that it's compiled together with the controller in order to get better performance.
- 6) External API. The main idea behind the layer is to provide language independent way to communicate with controller. This common example is the web-based RESTful API.
- 7) External network applications. Applications in any language leveraging services via External API exposed by controller services and internal applications. These applications are not needed in good performance and low latency communication with the controller. The common example is monitoring applications.
- 8) Web UI layer. It provides WEB-based user interface to manage the controller by setting up different parameters.

Also, the most important general question before choosing the controller or creating new one is what programming language to use. There is a trade off between the performance and the usability. For instance, POX controller written on Python is good for fast prototyping but it is too slow for production.

IV. EXPERIMENTAL CONTROLLERS EVALUATION

We performed an experimental evaluation of the controllers.

Our test bed consisted of two servers connected via 10Gb link. The first server was used to launch the controllers. The second server was used for traffic generation according to a certain test scenario.

We chose the following seven SDN/OpenFlow controllers:

- NOX [12] is a multi-threaded C++-based controller written on top of Boost library.
- POX [13] is a single-threaded Python-based controller. It's widely used for fast prototyping of network application in research.
- Beacon [14] is a multi-threaded Java-based controller that relies on OSGi and Spring frameworks.
- Floodlight [15] is a multi-threaded Java-based controller that uses Netty framework.
- MUL [16] is a multi-threaded C-based controller written on top of libevent and glib.
- Maestro [18] is a multi-threaded Java-based controller that uses JAVA.NIO library.
- Ryu [19] is Python-based controller that uses gevent wrapper of libevent.

Each controller runs the L2 learning switching application provided by the controller. There are several reasons for that. It's quite simple and at the same time representative.

It fully uses controller's internal mechanisms, and it also shows how effective the chosen programming language is by implementing single hash lookup.

We used the latest available sources of all controllers dated March, 2013. We run all controllers with the recommended settings for performance and latency testing, if available.

As a traffic generators we used freely available cbench [17] and our-own framework hcrprobe for controllers testing. Cbench and hcrprobe emulates any number of OpenFlow switches and hosts. Cbench is intended for measuring different performance aspects of the controller including the minimum and maximum controller response time, maximum throughput. Hcrprobe allows to investigate various characteristics of controllers in a more flexible manner by specifying patterns for generating OpenFlow messages (including malformed ones), varying the number of reconnection attempts in case the controller accidentally closes the connection, choosing traffic profile, and etc. It is written in Haskell that is high-level programming language and allows users to easily create their own scenarios for controllers testing.

Our testing methodology includes performance and scalability measurements as well as advanced functional analysis such as reliability and security. The goal of performance/scalability measurements is to obtain maximum throughput (number of outstanding packets, flows/sec) and minimum latency (response time, ms) for each controller. For reliability we measured the number of failures during long term testing under a given workload profile. And as for security we study how controllers work with malformed OpenFlow messages.

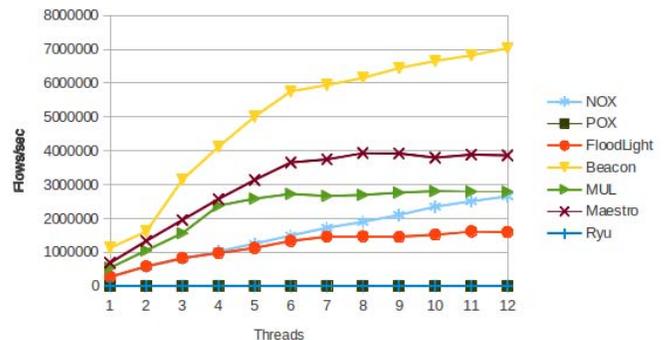


Fig. 4. The average throughput achieved with different number of threads.

The figure 4 shows the maximum throughput for different number of available cores per one controller. The single threaded controllers (Pox and Ryu) show no scalability across CPU cores. The performance of multithreaded controllers increases steady in line for 1 to 6 cores, and much slower for 7-12 cores because of using hyper threading technology (the maximum performance benefit of the technology is 40%). Beacon shows the best availability, achieving the throughput near 7 billion flows per second. This is because of using shared queues for incoming messages and batching for outgoing messages.

The average response times of all controllers are between 80-100ms. The long-term tests show that most controllers

when running for quite a long time start to drop connections with the switches and loose PacketIn messages. The average number is 100 errors for 24 hours. And almost all controllers crashes or losing the connection with a switch when they received malformed messages.

Let us come back to throughput numbers and understand if the current performance enough. In the data centers new flow request arrives every 10us in maximum and 300us to 2ms in average [22]. Assuming small data center with 100K hosts and 32 hosts/rack, the maximum flow arrival rate can be up to 300M with the median rate between 1.5M and 10M. Assuming 2M flows/sec throughput for one controller, it requires only 1-5 controllers to process the median load, but 150 for peak load! In large-scale networks the situation can be tremendously worse.

The solving of the problem should go two ways. The first way is improving single controller itself by doing more advanced multi-threaded optimizations. The second way is using multiple controller instances which collaboratively manage the network. This approach is called a distributed controller.

V. MOVING TO DISTRIBUTED CONTROLLER

As we see in previous section single controller is not enough for managing the whole network. There are two problems here:

- 1) *Scalability*. Because networks are growing rapidly, the controller's resources are not enough to maintain state of all network devices. Moreover, the flow setup latency in a bigger networks is also increasing.
- 2) *Reliability*. The controller is a single point of failure. If the controller crashes, the network stops.

To solve the above problems, we need physically distributed control plane with centralized view of the entire network.

The scheme of the solution is presented in Figure 5.

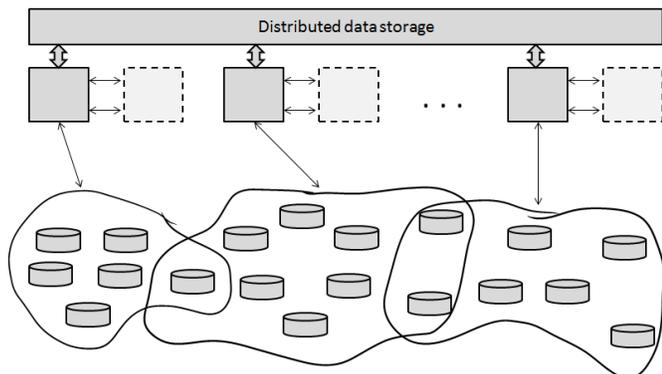


Fig. 5. The organization scheme of distributed controller.

The networks divides into segments, which controlled by dedicated instance of the controller. Network segments may overlap to ensure network resiliency in case of failure of any controller. In this case the switches will be redistributed over appropriate instances of the controller.

Each controller is connected to a distributed data storage that provides a consistent view of whole network. It stores all switch- and application- specific information. Application state is kept in the distributed data store to facilitate switch migration and controller failure recovery.

In addition, each controller has failover controller in case of its failure. It might be cold or hot. The cold failover is turned off by default and starts only when the master controller crashes. The hot failover receives the same messages as the master controller, but has read-only access. This provides the smallest recovery time.

There is a lot of open research questions like how to organized controllers consistency in the right way, how to reduce overhead on using distributed data store, how to do switch migration, how to run applications on distributed controllers, what the best controllers placement is, and etc.

VI. CONCLUSION

Software Defined Networking (SDN) has been developed rapidly and is now used by early adopters such as data centers. It offers immediate capital cost savings by replacing proprietary routers with commodity switches and controllers; computer science abstractions in network management offer operational cost savings, with performance and functionality improvements too.

However there is a lot researching has to be done especially in SDN software area. Controllers are not yet ready to use in production because of insufficient performance to operate with data centers and large scale networks load.

Distributed controller is the next step in developing SDN/Openflow controllers. It's solving the scalability and reliability problems of modern controllers. For this we must use the techniques already existed in software engineering.

REFERENCES

- [1] M. Casado, T. Koponen, D. Moon, and S. Shenker, *Rethinking Packet Forwarding Hardware*. In Proc. of HotNets, Nov. 2008.
- [2] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, Scott Shenker, *Rethinking enterprise network control*, IEEE/ACM Transactions on Networking (TON), v.17 n.4, p.1270-1283, August 2009
- [3] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, Scott Shenker. *Ethane: Taking Control of the Enterprise*, ACM SIGCOMM 07, August 2007, Kyoto, Japan.
- [4] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, Jonathan Turner, *OpenFlow: enabling innovation in campus networks*, ACM SIGCOMM Computer Communication Review, v.38 n.2, April 2008
- [5] J. Hamilton, *Data center networks are in my way*, Talk at Stanford Clean Slate CTO Summit, 2009.
- [6] M. Casado, T. Koponen, R. Ramanathan, S. Shenker S. *Virtualizing the Network Forwarding Plane*. In Proc. PRESTO (November 2010)
- [7] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, *Extending Networking into the Virtualization Layer*, HotNets-VIII, Oct. 22-23, 2009
- [8] J. Pettit, J. Gross, B. Pfaff, M. Casado, S. Crosby, *Virtual Switching in an Era of Advanced Edges*, 2nd Workshop on Data Center Converged and Virtual Ethernet Switching (DC-CAVES), ITC 22, Sep. 6, 2010
- [9] Open Networking Foundation, <https://www.opennetworking.org>
- [10] Openflow, <http://www.openflow.org>
- [11] Openflow specification, <http://www.openflow.org/wp/documents>

- [12] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. *NOX: towards an operating system for networks*. SIGCOMM Computer Communication Review 38, 3 (2008), 105-110.
- [13] Pox documentation, <http://www.noxrepo.org/pox/about-pox/>
- [14] Beacon documentation, <https://openflow.stanford.edu/display/Beacon/Home>
- [15] Floodlight documentation, <http://floodlight.openflowhub.org/>
- [16] Mul documentation, <http://sourceforge.net/p/mul/wiki/Home/>
- [17] Cbench documentation, <http://www.openflow.org/wk/index.php/Oflops>
- [18] Zheng Cai, *Maestro: Achieving Scalability and Coordination in Centralized Network Control Plane*, Ph.D. Thesis, Rice University, 2011
- [19] Ryu documentation, <http://osrg.github.com/ryu/>
- [20] Luigi Rizzo, *netmap: a novel framework for fast packet I/O*, Usenix ATC'12, June 2012
- [21] Packet Processing is Enhanced with Software from Intel DPDK, <http://intel.com/go/dpdk>
- [22] Theophilus Benson, Aditya Akella, and David A. Maltz, *Network traffic characteristics of data centers in the wild*, IMC, 2010

The Formal Statement of the Load-Balancing Problem for a Multi-Tenant Database Cluster with a Constant Flow of Queries

Evgeny A. Boytsov
Valery A. Sokolov
Computer Science Department
Yaroslavl State University
Yaroslavl, Russia
{boytsovea, valery-sokolov}@yandex.ru

Abstract — The concept of a multi-tenant database cluster offers new approaches in implementing a data storage for cloud applications. One of the most important questions to solve is finding a load-balancing algorithm to be used by the cluster, which is able to effectively use all available resources. This paper discusses theoretical foundations for such an algorithm in the simplest case when the flow of incoming queries is constant, that is, every tenant has a predefined intensity of the query flow and there are no changes in the state of the tenant's data.

Keywords — database; cluster; multi-tenancy; load-balancing; quadratic assignment problem; imitation modeling;

I. INTRODUCTION

When a company designs a high load cloud application, its developers sooner or later face the problem of organizing the storage of data in the cloud with the requirement of high performance, fault-tolerance and reliable tenants' data isolation from each other. At the moment these tasks are usually solved at the level of application servers by designing an additional layer of an application logic. Such a technique is discussed in many specialized papers for application developers and other IT-specialists [1, 2, 3]. There are also some projects of providing native multi-tenancy support at the level of a single database server [4]. This paper is devoted to an alternative concept of a multi-tenant database cluster which proposes the solution of the above problems at the level of a data storage subsystem and discusses theoretical foundations of this concept in a particular case.

II. THE ARCHITECTURE OF THE MULTI-TENANT DATABASE CLUSTER

A multi-tenant database cluster [5,6] is an additional layer of abstraction over ordinary relational database servers with a single entry point which is used to provide the isolation of cloud application customer data, load-balancing, routing of queries among servers and fault-tolerance. The main idea is to provide an application interface which has most in common with the interfaces of traditional RDBMS (relational database management system). At the moment the typical scenario of interaction with the cluster from the developer's point of view is seen as the following:

```
Connect( TenantId, ReadWrite / ReadOnly );  
SQL-commands  
Disconnect();
```

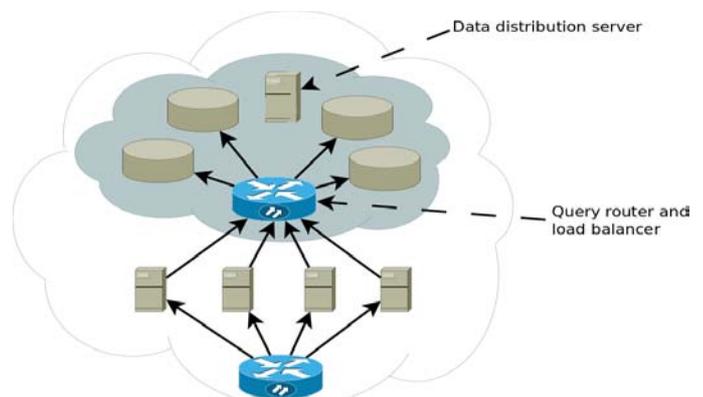


Fig. 1. Multi-tenant database cluster architecture

A multi-tenant cluster consists of a set of ordinary database servers and specific control and query routing servers.

The query routing server is a new element in a chain of interaction between application servers and database ones. This is the component application developers will deal with. In fact, this component of the system is just a kind of a proxy server which hides the details of the cluster structure, and whose main purpose is to find an executor for a query and route the query to him as fast as possible. It makes a decision based on the map of the cluster.

It is important to note that a query routing server has a small choice of executors for each query. If a query implies data modification, there is no alternative than to route it to the master database of a tenant, because only there data modification is permitted. If the query is read-only, it also can be routed to a slave server, but in the general case there would be just one or two slaves for a given master, so even in this case the choice is very limited.

The data distribution and load balancing server is the most important and complicated component of the system. Its main functions are:

- initial distribution of tenants data among servers of a cluster during the system deployment or addition of new servers or tenants;

- management of tenant data distribution, based on the collected statistics, including the creation of additional data copies and moving data to another server;
- diagnosis of the system for the need of adding new computing nodes and storage devices;
- managing the replication.

This component of the system has the highest value since the performance of an application depends on the success of its work.

III. MAIN CHARACTERISTICS OF THE QUERY FLOW

When modeling and analyzing the multi-tenant database cluster the most important things to study are characteristics and properties of the incoming query flow. The quality of implementation of this model component significantly affects the adequacy and applicability of results obtained during modeling.

The flow of incoming queries of the multi-tenant database cluster can be divided into N non-intersecting and independent sub-flows for each tenant $\lambda_i, i \in 1, N$:

$$A = \sum_{i=1}^N \lambda_i$$

The study of statistics on existing multi-tenant cloud applications shows that there is a significant dependency between the size of data that the client stores in the cloud and the intensity of a client's query flow (the more data has the client, the larger organization it represents and, therefore, the more often these data are accessed by individual users of that client). The analysis of the statistics also shows that the above tendency is not comprehensive and there are clients within the cluster which have the intensity of the query flow that does not match the size of the stored data (it can be both more or less intensive than it is expected). The client's query flow can be divided into two sub-flows: read-only queries and data-modifying ones.

$$\lambda_i = \lambda_{i_{read}} + \lambda_{i_{write}}$$

Such a division only has sense when the data replication is used within the cluster or when the solution is tuned to specifics of the particular database server or operating system. The higher-level analysis can omit this division.

Another obvious characteristics of the query flow is an average duration μ of a query at the server. The duration of different operations is not equal and this consideration should be taken into account during the modeling. This value has a significant impact on the quality of load-balancing, since it affects the formation of the total cluster load. As we know from the queuing theory, if $A\mu > N_{queries}$, where $N_{queries}$ is the maximum amount of queries that can run in parallel in the cluster, then the cluster will fail to serve the incoming flow of requests. It is also known that intensities of incoming query flows are changed during the lifetime of the application, that is, $\lambda_i = \lambda(t), i \in 1, N$. Some clients begin to use the application more intensively, the activity of others decreases, new clients appear and existing clients disappear. Besides, some applications may have season peaks of load.

IV. THE LOAD-BALANCING PROBLEM WITH A CONSTANT FLOW OF QUERIES

A. General problem

The present paper is devoted to the study of load-balancing the cluster in the case when flows of incoming queries have a constant intensity, i.e. $\lambda_i = const, i \in 1, N$. The solution of this problem can be considered as a solution of the general problem «at the point». Clusters without data replication will be studied (that is, without providing fault-tolerance). For simplicity we assume that $\mu = 1$ (or, equivalently, the bandwidth of each server in the cluster is divided by μ).

Let C be a multi-tenant database cluster that consists of M database servers (S_1, \dots, S_M) , for each of which we know the following values:

1. $\bar{\lambda}_i, i \in 1, \dots, M$ - the bandwidth of the database server;
2. $\bar{v}_i, i \in 1, \dots, M$ - the capacity of the database server.

There are also N clients, comprising the set T , for each of which we also know two values:

1. $\lambda_j, j \in 1, \dots, N$ - the intensity of j -th client query flow;
2. $v_j, j \in 1, \dots, N$ - the data size of j -th client.

We will call the $M \times N$ matrix X a distribution matrix (of clients in the cluster), if X satisfies the following constraints and conditions:

1. $x_{i,j} = 1$ when data of the j -th client are placed at the i -th server and $x_{i,j} = 0$ otherwise;
2. $\forall j \in 1, \dots, N \exists ! i \in 1, \dots, M : x_{i,j} = 1$ - the data of each client are placed at the single server;
3. $\forall i \in 1, \dots, M \sum_{j=1}^N x_{i,j} v_j \leq \bar{v}_i$ - the total data size at each server is less than or equal to the server capacity;
4. $\forall i \in 1, \dots, M \sum_{j=1}^N x_{i,j} \lambda_j \leq \bar{\lambda}_i$ - the total query flow intensity at each server is less than or equal to the server bandwidth.

We will call the matrix \tilde{X} the optimal matrix of distribution of clients set T in the cluster C if for some function $f(C, T, X)$ the following condition is met:

$$f(C, T, \tilde{X}) = \min\{f(C, T, X) : X - \text{distribution matrix}\}$$

The function f in this definition is the measure of load-balancing efficiency among the servers of the cluster. The problem of effective cluster load-balancing in this formulation reduces to finding an optimal distribution matrix \tilde{X} for a given cluster C , a set of clients T and the measure of efficiency f .

B. The measure of efficiency

What is the best way to measure the efficiency of load-balancing among servers? Uniformity of the load is a good criteria here, therefore the target function which will measure this characteristics should be searched. At the first approximation, it may seem that the sum of squares of differences between the load of the i -th server and the average load of servers in the cluster can be used as the above measure. This can be written as:

$$\sum_{i=1}^M \left(\frac{\sum_{j=1}^N x_{i,j} \lambda_j}{\bar{\lambda}_j} - Z \right)^2$$
, where Z — is the average load of the cluster servers, that is

$$Z = \frac{\sum_{i=1}^M \sum_{j=1}^N x_{i,j} \lambda_j}{M}$$

However, on closer examination this measure is consistent only if the cluster C consists of servers with a uniform performance, otherwise it leads to an intuitively wrong result. This consideration can be illustrated by the following example. Let the cluster C consist of twelve servers and two of them are 45 times more powerful than other ten (that is $\lambda_{1,2} = 45 \bar{\lambda}_k, k \in 3, \dots, 12$). In this case these two servers constitute 90 percent of total cluster computational power and therefore they play a crucial role in solving the problem of effective load-balancing. The operation mode of other ten servers is not important in such a configuration (for example, they can not serve any queries at all). But the above formula assumes that all terms are equivalent and therefore these ten servers will bring a decisive contribution to the measure. This example shows the need to somehow normalize the terms in accordance with the powers of the cluster components. So the desired situation can be formulated in the following way: the share of a total query flow at each server should be as close as possible to the share of this server in the total computational power of the entire cluster. Using this formulation, the function f can be written as follows:

$$f(C, T, X) = \sum_{i=1}^M \left(\frac{\sum_{j=1}^N x_{i,j} \lambda_j}{\sum_{j=1}^N \lambda_j} - \frac{\bar{\lambda}_i}{\sum_{i=1}^M \bar{\lambda}_i} \right)^2 \quad (1)$$

C. Additional considerations

Since a set of distribution matrices X is discrete and finite, then, if it is not empty (that is there are some feasible cluster configurations), there is a non-empty subset X_{min} , whose elements are the points of minimum of the target measure function f , that is, the problem of optimal cluster load-balancing always has a solution.

V. THE LOAD-BALANCING PROBLEM AS THE QUADRATIC ASSIGNMENT PROBLEM

The above problem is a special case of the generalized quadratic assignment problem (GQAP) which, in turn, is a

generalization of the quadratic assignment problem (QAP), initially stated in 1957 by Koopmans and Beckmann[7] to model the problem of allocating a set of n facilities to a set of n locations while minimizing the quadratic objective function arising from the distance between the locations in combination with the flow between the facilities. The GQAP is a generalized problem of the QAP in which there is no restriction that one location can accommodate only a single equipment. Lee and Ma[8] proposed the first formulation of the GQAP. Their study involves a facility location problem in manufacturing where facilities must be located among fixed locations, with a space constraint at each possible location. The aim is to minimize the total installation and interaction transportation cost. The formulation of the GQAP is:

$$\min \sum_{i=1}^M \sum_{j=1}^N \sum_{k=1}^M \sum_{n=1}^N f_{ik} d_{jn} x_{ij} x_{kn} + \sum_{i=1}^M \sum_{j=1}^N b_{ij} x_{ij}$$

subject to:

$$\sum_{i=1}^M s_{ij} \leq S_j, j \in 1, N,$$

$$\sum_{i=1}^M x_{ij} = 1, i \in 1, M,$$

$$x_{ij} \in \{0, 1\}, i \in 1, M, j \in 1, N,$$

where:

M is the number of facilities,

N is the number of locations,

f_{ik} is the commodity flow from a facility i to a facility k ,

d_{jn} is the distance from a location j to a location n ,

b_{ij} is the cost of installing a facility i at a location j ,

s_{ij} is the space requirement if a facility i is installed at a location j ,

S_j is the space available at a location j ,

x_{ij} is a binary variable which is equal to 1 if a facility i is installed at a location j .

The objective function sums the costs of installation and quadratic interactivity. The knapsack constraints impose space limitations at each location, and the multiple choice constraints ensure that each facility is to be installed at exactly one location.

The QAP is well known to be NP-hard [9] and, in practice, problems of moderate sizes, such as $n=16$, are still being considered very hard. For recent surveys on QAP, see the articles by Burkard [10], and Rendl, Pardalos, Wolkowicz [11]. An annotated bibliography is given by Burkard and Cela [12]. The QAP is a classic problem that defies all approaches for its solution and where the problems of dimension $n=16$ can be considered as of large scale. Since GQAP is a generalization of QAP, it is also NP-hard and even more difficult to solve.

The above load-balancing problem for a multi-tenant database cluster deals with hundreds of database servers and hundreds of thousands of clients. Due to NP-hardness of the

GQAP, it is obvious that such a problem can not be solved exactly or approximately with a high degree of exactness by an existing algorithm. So we can conclude that to solve the load-balancing problem we need to suggest some heuristics that can provide acceptable performance and measure its efficiency and positive effect in comparison with other load-balancing strategies.

VI. LOAD-BALANCING ALGORITHM HEURISTICS AND ITS EXPERIMENTAL VERIFICATION

To test the above-mentioned target function used to evaluate the efficiency of multi-tenant database cluster load-balancing strategy, an experiment was conducted on the simulation model of the cluster. The structure of the cluster with N database servers of different bandwidth (N is a parameter of the experiment) was created by using the modeling environment. At the initial moment the cluster had no clients. The model of the query flow was configured so that it provided progressive registration of new clients at the cluster and due to it the corresponding increase of query flow intensity. Subsystems of the model which provide data size refreshing and recalculation of tenants activity coefficients were disabled. Since the above subsystems are responsible for the dynamism of the model, the resulting configuration corresponded to a cluster with constant intensities of incoming query flows. Since the computational power of the cluster is limited and the total intensity of incoming query flow constantly increases, it is obvious that the cluster will stop serving queries at some point in time. It is also obvious that if one load-balancing strategy allows to place more clients than another within similar external conditions, then this load-balancing strategy is more effective and should be preferred in real systems. The experiment was meant to compare the simple algorithm, that is based on the analysis of incoming query flows intensities and the minimization of the above target function, with other simple load-balancing strategies which do not take intensities into account at all. It should be mentioned, that the ratio between read-only and data-modifying queries is not important in this experiment, since data replication is not used.

Three load-balancing algorithms are used in the experiment. The first algorithm tries to balance the load of the cluster by balancing the size of data that are stored at each server according to the server bandwidth ratio. When deciding to host a new client on the server, this algorithm calculates the ratio of the total data size of clients that are hosted on the server to the bandwidth of the server for all servers in the cluster (the amount of stored data per the processor core), and selects a server with the minimal ratio (if there are several such servers, it randomly selects one of them). The algorithm takes into account only the servers that have enough free space to host a new client. In a pseudo-code this algorithm can be written as the following:

```
min_servers = {};
min_ratio = max_double();
for each s in S
    if datasize( s ) + sz > capacity( s )
        continue;
    ratio = num_clients( s ) / bandwidth( s );
    if ratio < min_ratio
        min_ratio = ratio;
        min_servers.clear();
```

```
if ratio == min_ratio
    min_servers.add( s );
return random( min_servers );
```

Here S denotes a set of database servers within the cluster, $min_servers$ is a set of servers with minimum amount of clients taking into account server bandwidth, sz is a data size of a new client. This algorithm will be referred to as “Algorithm 1”.

The second algorithm tries to balance the load of the cluster by balancing the amount of clients at each server according to the server bandwidth ratio. When deciding to host a new client on the server, this algorithm calculates the ratio of the number of clients that are hosted on the server to the bandwidth of the server for all servers in the cluster (the number of clients per the processor core), and selects the server with a minimal ratio (if there are several such servers, it randomly selects one of them). As the previous algorithm, this algorithm also takes into account only the servers that have enough free space to host a new client. In a pseudo-code this algorithm can be written as the following:

```
min_servers = {};
min_ratio = max_double();
for each s in S
    if datasize( s ) + sz > capacity( s )
        continue;
    ratio = datasize( s ) / bandwidth( s );
    if ratio < min_ratio
        min_ratio = ratio;
        min_servers.clear();
    if ratio == min_ratio
        min_servers.add( s );
return random( min_servers );
```

The meaning of variables here is the same as in the previous example. This algorithm will be referred to as “Algorithm 2”.

The third algorithm is based on the minimization of the target function (1). For the sake of simplicity this algorithm was connected to the query generator information subsystem of the model to get exact values of incoming query flow intensities for each client. In reality such an approach cannot be implemented and values of query flow intensities should be obtained by some statistical procedures, but for experimental purposes and testing theoretical model this approach is applicable. The main principle of the algorithm is simple: it alternately tries to host a new client at each server and computes the resulting value of the target function (1). Finally, the client is hosted at the server which gave the minimal value of all the above. In a pseudo-code this algorithm can be written as the following:

```
min_server = null;
min_ratio = max_double();
for each s in S
    if datasize( s ) + sz > capacity( s )
        continue;
    ratio = F( S | new client hosted at s );
    if ratio < min_ratio
        min_ratio = ratio;
        min_server = s;
return min_server;
```

In this example F denotes the target function (1). This algorithm will be referred to as “Algorithm 3”.

All three algorithms were tested in the same environment, that is, with the same mean of query cost and tenants activity coefficient distribution. The results of experiments are given in the table 1. The first three columns show the parameters of the model and the algorithm used in the particular experiment. The fourth column shows the average amount of clients hosted at the cluster when the model met the experiment stop condition (one of the servers had the queue with more than 200 pending requests). Algorithm number 3 has shown significantly better results than others for all three models.

TABLE I. RESULTS OF EXPERIMENTS

Number of servers	Algorithm	Number of experiments	Average number of hosted clients
5	1	30	701,95
5	2	30	1197,63
5	3	30	1353,45
9	1	30	1090,6
9	2	30	1851,7
9	3	30	2155,45
15	1	30	1766,5
15	2	30	3235,35
15	3	30	3835,2

VII. CONCLUSION

The experiment has shown that the load-balancing strategy based on the analysis of incoming query flow intensities is more effective than others. This fact leads to the conclusion that the above-mentioned theoretical concepts are correct and can be applied to construct more complicated load-balancing strategies which take into account more factors and can be used in a more complicated environment. Some interesting questions to study are:

- How to determine the incoming query flow intensity of a client in a real environment;

- What algorithms can be used to find a better solution for the clients assignment problem;
- Are all solutions of the client assignment problem equally valuable when the intensities of incoming query flows are not constant;
- How to deal with the data replication and how the intensity of a client query flow should be divided among servers which have copies of the client data;
- What strategy should be used to relocate the clients data when the load balancing subsystem decides to do so.

All these questions are crucial in implementing an efficient load-balancing strategy for the cluster.

- [1] F. Chong, G. Carraro, "Architecture Strategies for Catching the Long Tail", Microsoft Corp. Website, 2006.
- [2] F. Chong, G. Carraro, R. Wolter, "Multi-Tenant Data Architecture", Microsoft Corp. Website, 2006.
- [3] K.S. Candan, W. Li, T. Phan, M. Zhou, "Frontiers in Information and Software as Services", proceedings of ICDE, 2009, pages 1761-1768.
- [4] Oliver Schiller, Benjamin Schiller, Andreas Brodt, Bernhard Mitschang, "Native Support of Multi-tenancy in RDBMS for Software as a Service", proceedings of the 14-th International Conference on Extending Database Technology, 2011.
- [5] E.A. Boytsov, V.A. Sokolov, "The Problem of Creating Multi-Tenant Database Clusters", proceedings of SYRCoSE 2012, Perm, 2012, pages 172-177.
- [6] E.A. Boytsov, V.A. Sokolov, "Multi-tenant Database Clusters for SaaS", proceedings of BMSD 2012, Geneva, 2012, page 144.
- [7] Koopmans, T.C. and M.J. Beckmann, "Assignment problems and the location of economic activities", *Econometrica* 25, 1957, pages 53-76.
- [8] Lee C.-G. and Z. Ma, "The generalized quadratic assignment problem", Research Report, Department of Mechanical and Industrial Engineering, University of Toronto, Toronto, Canada, 2004.
- [9] S. Sahni and T. Gonzales, "P-complete approximation problems", *Journal of ACM*, 1976.
- [10] R.E. Burkard, "Locations with spatial interactions: the quadratic assignment problem", *Discrete Location Theory*. John Wiley, 1991.
- [11] P. Pardalos, F. Rendl, and H. Wolkowitz. "The quadratic assignment problem: A survey and recent developments", proceedings of the DIMACS Workshop on Quadratic Assignment Problems, volume 16 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 1-41, 1994.
- [12] R.E. Burkard and E. Cela, "Quadratic and three-dimensional assignment problems", Technical Report SFB Report 63, Institute of Mathematics, University of Technology Graz, 1996.

Scheduling signal processing tasks for antenna arrays with simulated annealing

Daniil A. Zorin

Department of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, Russia
juan@lvk.cs.msu.su

Abstract — The problem dealt with in this paper is the design of a parallel embedded system with the minimal number of processors. The system is designed to solve signal processing tasks using data collected from antenna arrays. Simulated annealing algorithm is used to find the minimal number of processors and the optimal system configuration.

Keywords — optimization, scheduling, hardware design, embedded systems, simulated annealing

I. INTRODUCTION

Antenna array is hardware used to collect data from the environment, it is often employed in areas such as radiolocation and hydroacoustics [1]. Radiolocation tools have to process signals with a fixed frequency and have hard deadlines for the data processing time. At the same time, the size of the antenna array is limited, therefore, in order to maintain high accuracy, algorithms with significant computational cost have to be used to process signals. The co-design problem of finding the minimal necessary number of processors and scheduling the signal processing tasks on it arises in this relation. This paper suggests the application of simulated annealing algorithm to this problem. The purpose of this work is to show how the simulated annealing algorithm (discussed, for instance, in [2] and [3]) can work with real-world industrial problems.

In Section 2 we define the problem of scheduling for systems with antenna arrays and show the structure of signal processing algorithms used in such systems. It is explained how this problem can be formulated in the way that allows to use simulated annealing. The simulated annealing algorithm itself is discussed in Section 3. Experimental results obtained with the algorithm are given in Section 4.

II. PROBLEM FORMULATION

Systems used in radiolocation and hydroacoustics use a set of sensors to collect data from the environment. This set is called antenna array, and it is the most valuable and complicated part of the system. The size of the antenna array is fixed, so it is preferable and cheaper to build the system with a smaller antenna array and effective performance.

The problem solved with the antenna array system is finding the coordinates of the source of the signal. Traditional signal processing algorithms are based on fast Fourier

transforms (FFT). However, their potential solution capabilities are limited by the sizes of the antenna array. With a small array, the FFT will be performed on a small set of points, which can lead to low accuracy. Alternative methods based on automatic interference filtration [4] and on correlation matrix expansion (also shortened to CME) [5] can give accurate results even with smaller antenna arrays. They use several samples collected with the small array over a period of time to get very precise solutions. However, their computational complexity is significantly higher.

Assume that the antenna array has K elements (sensors) and works in frequency interval $(-B, B)$. The interval is split into L parts, and each of them is processed separately until the final result is computed on the last stage of the algorithm. We also need the number of support vectors used in CME method, M_0 . The sampling frequency is $1/\tau = aB$, where $a \geq 2.5$ is the coefficient in the Kotelnikov-Nyquist theorem. The system waits until a sample of n points is collected from the sensors, and then starts the processing algorithm. Therefore, the execution deadline is n/τ , the time until the new sample is collected. If the deadline is broken, the sensors' buffer will overflow.

Stage	Name	Complexity	Input size	Output size
1	Normalization	$O(aL)$	-	aL
2	FFT	$O(aL \cdot \log_2 aL)$	aL	1
3	Vector multiplication	$O(K^2)$	1	K^2
4	Computing eigenvalues, matrix reversal	$O(K^3)$	K^2	K^2
5	Computing signal source coordinates	$O(K)$	K^2	K
6	Vector multiplication	$O(K)$	K	1
7	Vector comparison	$O(K)$	1	K
8	Vector comparison	$O(LK)$	K	-

Table 1. Steps of the CME method

The steps of the algorithm, their respective computational complexities and the size of the data processed on each step are shown in Table 1. For simplicity, all figures are given only for the CME method. However, the general scheme of the automatic filtration method is the same, the difference lies on step 5, where the complexity becomes $O(K^2)$.

The signal processing runs on a multiprocessor system. It is assumed that processors are identical. Processors have fixed clock rate and reliability. The processors are interconnected, data transfer rate is fixed.

The workflow of the program is shown in Figure 6. The nodes represent subprograms and the edges represent dependencies between them. First, preprocessing, including FFT, is applied to the collected data, and the corresponding nodes are in the topmost row. They implement steps 1-3 from Table 3. The number of nodes is the same as K , the size of the antenna array. Then all data is sent to each of the L subprocesses and CME is performed. The CME can be divided into steps; each step implies some operations on the matrix (nodes $CME_i_stage_j$, where i runs from 1 to L , and j enumerates the stages). In Figure 6, there are three stages that correspond to steps 4 and 5 in Table 1. In the latter half the CME is broken into M_0 parallel threads for step 6 and joined into one thread on step 7 (nodes $CME_i_pstage_j_k$, where i runs from 1 to L , j runs from 1 to M_0 , and k enumerates the stages). Then all data is collected for final processing on step 8 (CME_final).

All nodes perform simple computations with matrices and vectors, such as FFT or matrix multiplications, so the complexity of each subprogram (also called ‘task’ hereafter) is known. Since processor performance is known, it is possible to calculate the execution time of each node, as well as the amount of data sent between the nodes. So, we reach the following mathematical problem statement [2].

The signal processing program can be represented with its data flow graph $G = \{V, E\}$, where V is the set of vertices (corresponding to the tasks) and E is the set of edges (corresponding to the dependencies of the tasks). Each vertex is marked by the time of execution of the corresponding task and each edge is marked by the time of data transfer. The set of processors denoted by M is given.

Processor redundancy implies adding a new processor to the system and using it to run the same tasks as on some existing processor. In this case the system fails if both processors fail. The additional processor is used as hot spare, i.e. it receives the same data and performs the same operations as the primary processor, but sends data only if the primary one fails. With switch architecture used, this does not cause any delays in the work of the system.

A schedule for the program is defined by task allocation, the correspondence of each task with one of the processors, and task order, the order of execution of the task on the processor. Formally, a schedule is defined as a pair (S, D) where S is a set of triplets (v, m, n) where $v \in V$, $m \in M$, $n \in \mathbb{N}$, so that $\forall v \in V : \exists ! s = (v_i, m_i, n_i) \in S : v_i = v$; and $\forall s_i = (v_i, m_i, n_i) \in S, \forall s_j = (v_j, m_j, n_j) \in S : (s_i \neq s_j \wedge m_i = m_j) \Rightarrow n_i \neq n_j$.

D is a multiset of elements of the set of processors. Substantially m and n denote the placement of the task on a processor and the order of execution for each version of each task. The multiset D denotes the spare processors: if processor m has k spares, it appears in D k times.

A schedule can be represented with a graph. The vertices of the graph are the elements of S . If the corresponding tasks are connected with an edge in the graph G , the same edge is added to the schedule graph. Additional edges are inserted for all pairs of tasks placed on the same processor right next to each other.

According to the definition, there can be only one instance of each task in the schedule, and all tasks on any processor have different numbers. Besides these, one more limitation must be introduced to guarantee that the program can be executed completely. A schedule S is correct by definition if its graph has no cycles. Otherwise the system would reach a deadlock where two processors are waiting for data from each other forever.

For every correct schedule the following functions are defined: $t(S)$ – time of execution of the whole program, $R(S)$ – reliability of the system, $M(S)$ – the number of processors used.

Given the program G , t_{dir} , the hard deadline of the program, and R_{dir} , the required reliability of the system, the schedule S that satisfies both constraints ($t(S) < t_{dir}$ and $R(S) > R_{dir}$) and requires the minimal number of processors is to be found.

Theorem 1. The optimization problem formulated above is NP-hard.

III. SIMULATED ANNEALING ALGORITHM

The proposed algorithm of solution is based on simulated annealing [6]. For simplicity, the model used in this study does not consider software reliability, so operations and structures related to that are omitted here. This does not affect the algorithm’s performance because it simply works as if the software reliability is always maximal.

The following three operations on schedules are used.

Add spare processor and *Delete spare processor*. Adds or removes a hot spare to the selected processor.

Move vertex. This operation changes the order of tasks on a processor or moves a task on another processor. It is obligatory to make sure that no cycles appear after this operation. The analytic form of the necessary and sufficient condition of the correctness of this operation is given in [4].

Theorem 2. If A and B are correct schedules, there exists a sequence of operations that transforms A to B such that all interim schedules are correct.

Each iteration of the algorithm consists of the following steps:

Step 1. Current approximation is evaluated and the operation to be performed is selected.

Step 2. Parameters for the operation are selected and the operation is applied.

Step 3. If the resulting schedule is better than the current one, it is accepted as the new approximation. If the resulting schedule is worse, it is accepted with a certain probability.

Step 4. Repeat from step 1.

The number of iterations of the algorithm is pre-determined.

If the reliability of the system is lower than required, spare processors and versions should be added, otherwise they can be deleted. If the time of execution exceeds the deadline the possible solutions are deleting versions or moving vertices. The selection of the operation is not deterministic so that the algorithm can avoid endless loops.

When the operation is selected, its parameters have to be chosen. For each operation the selection of its parameters is nondeterministic, however, heuristics are employed to help the algorithm move in the direction where the new schedule is more likely to be better.

The selection of the operation is not deterministic so that the algorithm can avoid endless loops. The reliability limit and the deadline can either be satisfied or not. Probability of selecting each operation, possibly zero, is defined for each of the four possible situations depending on the time and reliability constraints ($tdir$ and $Rdir$): both constraints satisfied, both constraints not satisfied, reliability constraint is satisfied while the time constraint is not, and vice versa. These probabilities are given before the start of the algorithm as its settings.

Some operations cannot be applied in some cases. For example, if none of the processors have spare copies it is impossible to delete processors and if all versions are already used it is impossible to add more versions. Such cases can be detected before selecting the operation, so impossible operations are not considered.

When the operation is selected, its parameters have to be chosen.

Add spare processor. Processors with fewer spares have higher probability of being selected for this operation.

Delete spare processor. A spare of a random processor is deleted. The probability is proportional to the number of spare processors.

The probabilities for these operations are set with the intention to keep balance between the reliability of all components of the system.

Move vertex. If $t(S) < tdir$, the main objective is to reduce the number of processors. With a probability of $pcut$ the following operation is performed: the processor with the least tasks is selected and all tasks assigned to it are moved to other processors. With a probability of $1-pcut$ the movement of a task is decided by one of the three strategies described below.

If $t(S) > tdir$, it is necessary to reduce the time of execution of the schedule. It can be achieved either by moving several tasks to a new processor or reallocating some tasks. The

parameters for the operation are chosen according to one of the three strategies: delay reduction, idle time reduction or mixed.

Delay reduction strategy. The idea of this strategy emerges from the assumption that if the time of the start of each task is equal to the length of the critical path to this task in graph G, the schedule is optimal. The length of the critical path is the sum of the lengths of all the tasks forming the path and it represents the earliest time when the execution of the task can begin.

For each element s it is possible to calculate the earliest time when s can start, i.e. when all the tasks preceding the current one are completed. The difference between this time and the moment when the execution of s actually starts according to the current schedule is called the delay of task s . If some task has a high delay, it means that some task preceding it is blocking its work, so the task before the one with a high delay has to be moved to another processor.

The task before the task with the highest delay is selected for Move Vertex operation. If the operation is not accepted, on the next iteration the task before the task with the second highest delay is selected, and so on. The position (pair (m, n) from the triplet) is selected randomly among the positions where the task can be moved without breaking the correctness condition.

Figure 1 gives an example of delay reduction. Task 3 does not depend on task 4, so moving task 4 to the first processor reduces the delay of task 3, and the total time decreases accordingly.

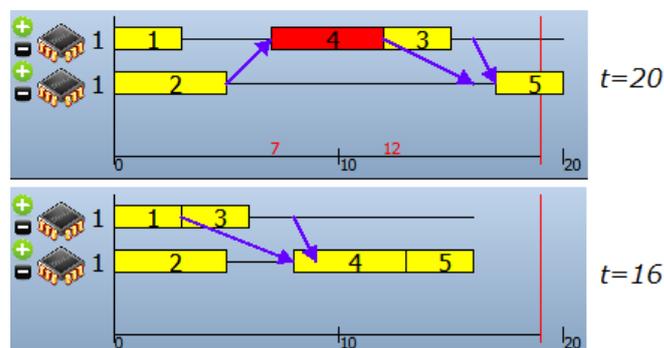


Figure 1. Delay reduction strategy

Idle time reduction strategy. This strategy is based on the assumption that in the best schedule the total time when the processors are idle and no tasks are executed due to waiting for data transfer to end is minimal.

For each position (m, n) the idle time is defined as follows. If $n=1$ then its idle time is the time between the beginning of the work and the start of the execution of the task in the position $(m, 1)$. If the position (m, n) denotes the place after the end of the last task on the processor m , then its idle time is the time between the end of the execution of the last task on m and the end of the whole program. Otherwise, the idle time of the position (m, n) is the interval between the end of the task in $(m, n-1)$ and the beginning of the task in (m, n) .

The task to move is selected randomly with higher probability assigned to the tasks executed later. Among all positions where it is possible to move the selected task, the position with the highest idle time is selected. If the operation is not accepted, the position with the second highest idle time is selected, and so on.

The idle time reduction strategy is illustrated in Figure 2. The idle time between tasks 1 and 4 is large and thus moving task 3 allows reducing the total execution time.

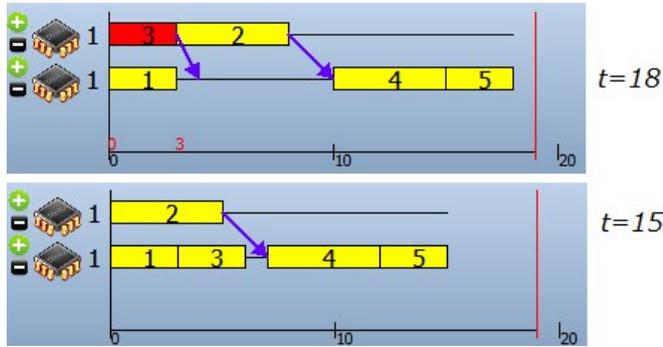


Figure 2. Idle time reduction strategy

Mixed strategy. As the name suggests, the mixed strategy is a combination of the two previous strategies. One of the two strategies is selected randomly on each iteration. The aim of this strategy is to find parts of the schedule where some processor is idle for a long period and to try moving a task with a big delay there, prioritizing earlier positions to reduce the delay as much as possible. This strategy has the benefits of both idle time reduction and delay reduction, however, more iterations may be required to reach the solution.

After performing the operation a new schedule is created and time, reliability and number of processors are calculated for it. Depending on the values of these three functions the new schedule can be accepted as the new approximation for the next iteration of the algorithm. Similar to the standard simulated annealing algorithm, parameter d modeling the temperature is introduced. Its initial value is big and it decreases after each iteration.

The probability to accept a worse schedule on step 3 depends on the parameter called temperature. This probability decreases along with the temperature over time. Temperature functions such as Boltzmann and Cauchy laws [7] can be used as in most simulated annealing algorithms

Theorem 3. If the temperature decreases at logarithmic rate or slower, the simulated annealing algorithm converges in probability to the stationary distribution where the combined probability of all optimal results is 1.

IV. EXPERIMENTS

Figure 7 shows the solution found by the algorithm for the problem shown on Figure 6. The system has been successfully reduced to 4 processors.

In real systems the size of the array is a power of 2, usually between 256 and 1024 (radiolocation systems use smaller

arrays), and the number of frequency intervals (L) is a power of 2, usually 32 or 64. For evaluation purposes, other values of L were tested as well. The value of M_0 is normally between 1 and 4.

In general, the majority of computations are performed after the initial processing on the K antennas and constitute the $L \cdot M_0$ parallel sequences of nodes in the program graph. Therefore, the quality of the algorithm can be estimated by comparing the number of processors in the result with the default system configuration where $L \cdot M_0$ processors are used. The following graphs (Figures 3-5) show the quotient of these two numbers, depending on L , for radiolocation problem. Lower quotient means better result of the algorithm.

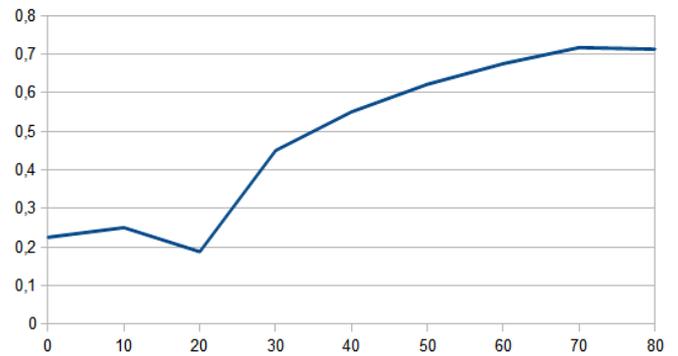


Figure 3. Optimization rate, $M_0=2$

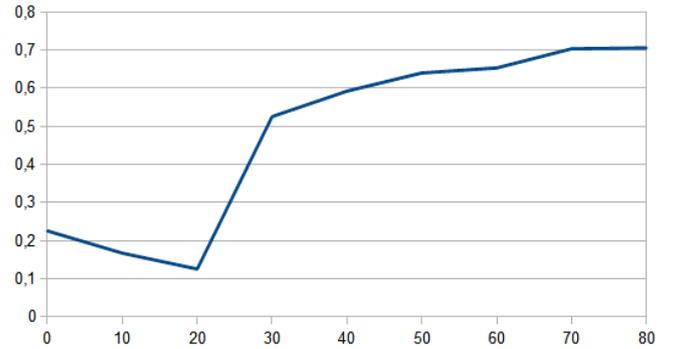


Figure 4. Optimization rate, $M_0=3$

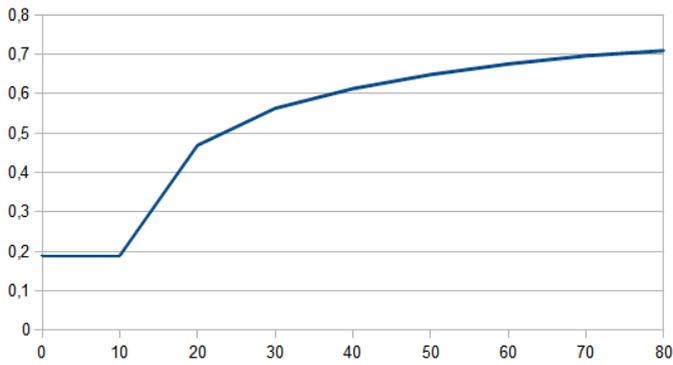


Figure 5. Optimization rate, $M_0=4$

As we can see, the algorithm optimizes the multiprocessor system by at least 25% in harder examples with many parallel tasks, and by more than a half in simpler cases.

CONCLUSIONS

Experiments with our tool testify that scheduling for antenna arrays can be done effectively with simulated annealing. The experimental data shows that the size of the

system can be optimized by 25-30% without breaking deadlines and limits of reliability.

REFERENCES

- [1] Kostenko V.A. Design of computer systems for digital signal processing based on the concept of "open" architecture. //Automation and Remote Control. – 1994. – V. 55. – №. 12. – P. 1830-1838.
- [2] D. A. Zorin and V. A. Kostenko Algorithm for Synthesis of Real-Time Systems under Reliability Constraints // Journal of Computer and Systems Sciences International. 2012. Vol. 51. No. 3. P. 410–417.
- [3] Daniil A. Zorin, Valery A. Kostenko. Co-design of Real-time Embedded Systems under Reliability Constraints // Proceedings of 11th IFAC/IEEE International Conference on Programmable Devices and Embedded Systems (PDeS). Brno, Czech Republic: Brno University of Technology, 2012. P. 392-396
- [4] Monzingo R. A., Miller T. W. Introduction to adaptive arrays, 1980 //Wiley New York. – P. 56-63.
- [5] Widrow B., Stearns S. D. Adaptive signal processing //Englewood Cliffs, NJ, Prentice-Hall, Inc., 1985, 491 p. – 1985. – V. 1.
- [6] Kalashnikov, A.V. and Kostenko, V.A. (2008). A Parallel Algorithm of Simulated Annealing for Multiprocessor Scheduling. Journal of Computer and Systems Sciences International, 47, No. 3, pp. 455-463.
- [7] Wasserman F. Neurocomputer Techniques: Theory and Practice [Russian translation] //Mir, Moscow. – 1992. – 240 p.

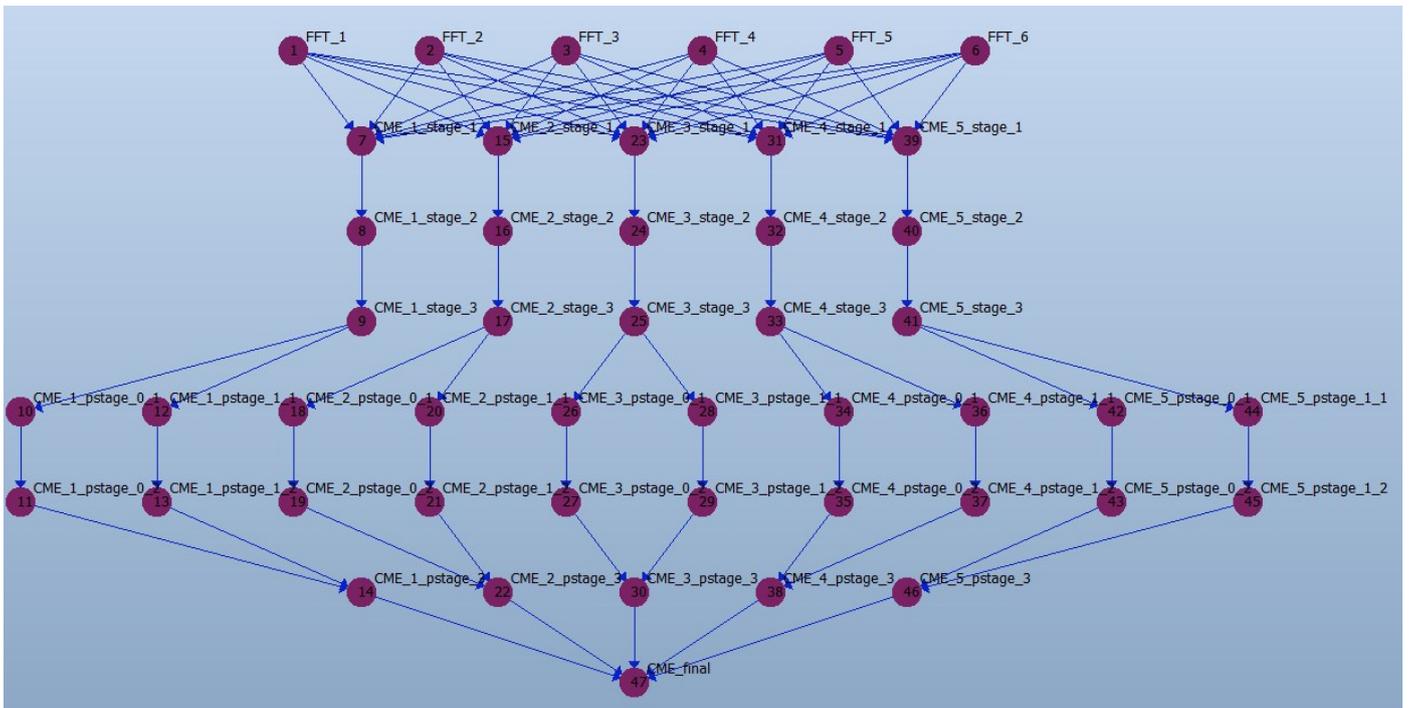


Figure 6. Signal processing workflow

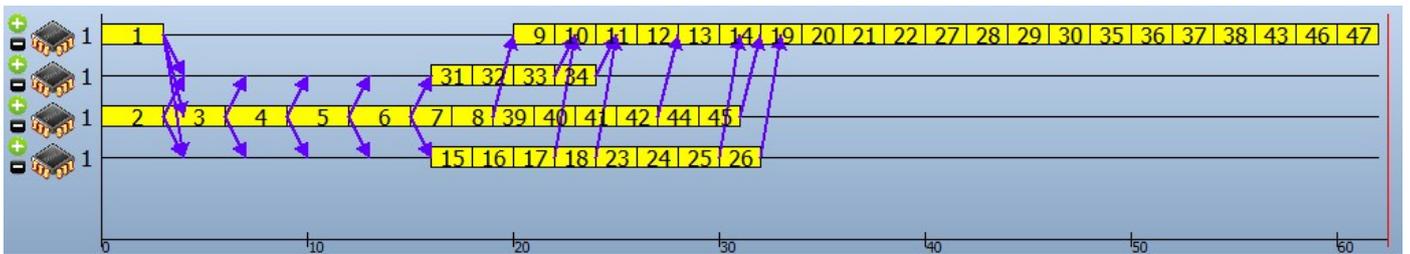


Figure 7. Schedule for the program from Figure 6

Automated deployment of virtualization-based research models of distributed computer systems

Andrey Zenzinov

Mechanics and mathematics department, Moscow State University

Institute of mechanics, Moscow State University

Moscow, Russia

andrey.zenzinov@gmail.com

Abstract—In the research and development of distributed computer systems and related technologies it is appropriate to use research models of such systems based on the virtual infrastructure. These models can simulate a large number of different configurations of distributed systems. The paper presents an approach to automate the creation of virtual models for one of the classes of distributed systems used for scientific computing. Also there considers the existing ways to automate some maintaining processes and provides practical results obtained by the author in the development and testing of prototype software tools to create virtual models.

Index Terms—Distributed computer systems, Virtualization, Automation, Grid computing

I. INTRODUCTION

Research and development of distributed computer systems usually involves performing a lot of testing and development activities. It seems useful to carry out experiments and tests not on a production system, but on its research model created specifically for the purposes of performing the experiments. Virtualization-based research models of computer systems may be used to accurately model software components of such systems. This kind of models is widely used in the study of distributed systems [1], [2].

Another possible use case for virtual research models is development of parallel and distributed programs, e.g. client-server applications, parallel computing programs. It is also applicable to information security sphere, particularly in the development of different monitoring and auditing systems.

With virtualization technologies it is possible to simulate distributed systems of various architecture. Also, the virtualization-based approach significantly simplifies the process of deploying the model and preparing the experiments.

The main idea of this approach is to use one or more computers (virtualization hosts) with a set of deployed virtual machines which run the software identical or similar to the software in the production system. Similar approaches are widely used, e.g. in cloud computing, to deploy multiple computing nodes on a single physical host. The overhead of running a set of virtual machines is relatively low on hosts with hardware virtualization support.

In this research we consider grid computing systems designed for parallel task execution as the object of modelling. Typical examples of these systems are the distributed systems based on the Globus Toolkit—a set of open source software

used for building grid computing systems. Distributed systems of this kind usually do not require the use of virtualization technologies to function, contrary to other types of distributed systems, e.g. in cloud computing. This property of typical grid computing systems simplifies the virtualization-based modelling as nested virtualization is not required in this case.

In our research we consider evaluation of information security properties of distributed systems as the goal of modelling. The following attacks are particularly relevant to grid computing systems: denial of service (DoS) and distributed denial of service (DDoS) attacks; exploitation of software vulnerabilities; attacks on the system's infrastructure allowing the attacker to eavesdrop and to substitute trusted components of the system.

Different kinds of modelling parameters should be taken into account, such as the system's architecture, attacker location, configuration and composition of security mechanisms, and attack scenarios. Usually it is necessary to perform a series of experiments for each configuration of parameters to show the adequacy of the experiments' coverage. Modelling different variants of system's architecture requires to iteratively rebuild and redeploy the model, which may be performed at a high degree of automation when using the virtualization-based models.

II. WORKFLOW OF DISTRIBUTED SYSTEM DEPLOYMENT

The building process of the distributed system contains the following steps:

- a set of nodes creation;
- OS installation and setting up on each node;
- additional software installation on each node;
- setting up distributed network.

All these processes takes a lot of time and it is a very monotonous work, which requires carefulness and attention, because mistakes can lead to system failure.

Suppose the operator performing the deployment processes have got given system configuration, which describes nodes of the distributed system, its network architecture, a set of software tools installed on the nodes and other necessary options. The distributed system and its virtual model are being constructed following this configuration.

The operator should perform actions based on the algorithm, which was described above. Some of these actions require their completion, which can take a long time, e.g. disk image copying, software installation, etc. The operator can make mistakes, which may result in system performance loss.

It seems appropriate to reduce the amount of non-automated actions to increase the reliability of deployment.

III. GOALS OF THE RESEARCH

The aim of the study is to automate the deployment and setting up of virtual research model with given configuration file. Let us require following options from the deployment system:

- support for different types of nodes;
- nodes have a configured required software for remote access to nodes, e.g. via SSH;
- the deployment system should work only with open source software.

The idea of last requirement is that we may need to modify the code of some programs for further development. Different tasks in a distributed system determines different types of nodes. For example, we can divide grid nodes into several types: compute nodes, gateway nodes, certificate distribution nodes, task distribution nodes. These types have different software and configurations. On the other side usually there are not much kinds of nodes.

IV. WAYS TO AUTOMATION

Let us consider the process of deployment. It is divided into the steps, as described above, and we're using VMs to emulate nodes. Network infrastructure is also virtualized.

It is convenient to use libvirt [3] for virtual system management. Libvirt is an open source cross-platform API, daemon and management tool for managing platform virtualization. It provides unified controlling for most hypervisors like KVM, Xen, VMware and others. There are API for some popular programming languages like Python. The idea of automation lies in using libvirt, well-known shell scripting automation and programs written in Python.

OS installing is usually interactive. It contains a set of specific questions about disk partitions, packages, users, time zone, etc. These questions are obstacles for automated OS installing. However, there are various solutions such as network installation and using specified file with the answers. These solutions have been successfully used in many modern systems, e.g. compatible with the Debian GNU/Linux and Red Hat Enterprise Linux.

Automation is also applicable to editing configuration files. On the one hand, this implements with text processing tools and specific actions using regular expressions. On the other hand, there are special systems designed to automate the configuration of OS and software – Chef and Puppet [4], [5].

V. RELATED WORKS

Automation of VM deployment is studied in IBM developerWorks paper “Automate VM deployment“ [6]. In this paper the author propose to create separate virtual machines on a given configuration. The described system consists of two parts: Virtual Machine Deployment Manager (VDM) and Virtual Machine Configuration Manager (VCM). Configuration for each VM stored on special disk image. Then it boots on the VM via virtual CD-ROM and configure the system. The VDM handles user requests to deploy VM such as cloning virtual images, configuring VM hardware settings, and registering the VM to the VM hypervisor. The VCM is installed in the VM template. After a system starts, it will start automatically and launch the configuration applications on the CD with configuration data.

The deployment process is illustrated on figure 1.

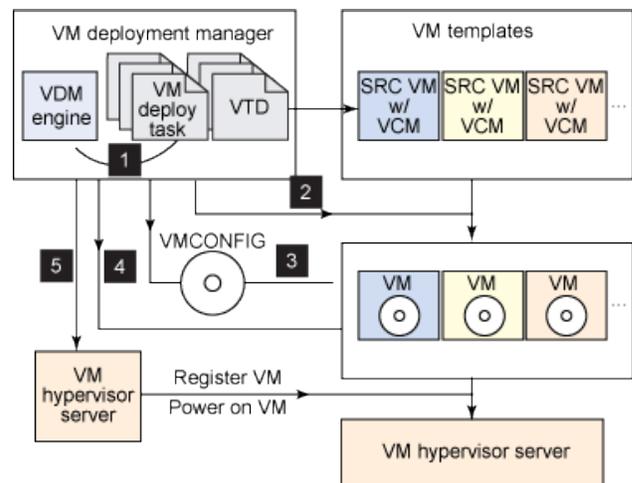


Fig. 1. Architecture of the automatic VM deployment framework

Using pre-configured VM templates and configuration files is the main advantage in this paper. This system is based on VMware virtualization and shell scripts. It supports Red Hat Enterprise Linux, SuSE Linux and Windows.

There are also some disadvantages. Unfortunately, described system does not support creating multiple copies of VM template, which is essential for our research. Use of special configuration CD seems redundant.

Another approach is presented in Vagrant system [7]. Vagrant is an open source tool for building development environment using virtual machines. The idea of this system is to use already prepared VM images, called “boxes“. You need only three commands:

```
vagrant box add <name> <box url>
vagrant init <name>
vagrant up
```

These commands launches pre-configured VM with specific configuration. You should create another box to

create a machine with different configuration. Configuration parameters stores in "vagrantfile" which describes machine settings such as hostname, network settings, SSH settings and provider (hypervisor) settings. VirtualBox is a default provider for Vagrant. But you can use other hypervisors like VMware via special plugins. Additional software can be installed using the Chef and Puppet.

This system is simple to use, there are Chef, Puppet, SSH, Network File System (NFS) support – it is a major advantage. There are multi-machine support: each machine describes separately in "vagrantfile". But concept using in Vagrant suppose that there is a "master" machine and limited number of "slave" machines. It is not convenient for large-scale distributed system.

VMware presents "Auto Deploy" technology in their products [8]. Auto Deploy is based on the Preboot eXecution Environment (PXE) – environment to boot computers using a network interface. Another important part is vSphere PowerCLI – a PowerShell based command line interface for managing vSphere.

Unfortunately, Auto Deploy deals only with VMware ESXi hypervisor and it only available in VMware vSphere Enterprise Plus version, which is non-free.

We should consider CFEngine. It is an open source configuration management system widely used for managing large numbers of hosts that run heterogeneous operating systems. There is support for a wide range of architectures: Windows (cygwin), Linux, Mac OS X, Solaris, AIX, HPUX, and other UNIX-systems. This system is not directly related to virtualization, but it is a proven tool for large systems. The main idea is to use unified configurations that describe required state of the system.

CFEngine automates file system operations, service management and system settings cloning.

VI. REQUIREMENTS FOR THE DEPLOYMENT SYSTEM

After the review of existing approaches to automation, let us formulate the requirements for the deployment system:

- using of general configuration;
- VM templates using;
- the ability to create a set of clones of the template VM;
- automated initialization;
- ability to making manual setting up.

Using of the general configuration assumes unified method of describing various systems with general parameters. It means that there is a unified set of parameters for all modelling systems. Requirements for configuration with these parameters are described below. If the simulated system contains of a large number of repetitive nodes, it seems appropriate to use a special VM templates. In this case you should make requested number of copies and possibly add

some changes to their configuration. There are two ways of cloning VMs: complete cloning – copying the template disk image; incremental cloning – the base image used in the "read-only" mode, and changes of clone's settings saved in separate files. The second way can significantly reduce disk space usage and deployment time. This economy is particularly noticeable in the large series of experiments. We should note that there is an analogue of this approach applied to memory. It is KSM (Kernel SamePage Merging) technology. There is also KSM modification – UKSM (Ultra KSM).

As to the requirement of ability to manual setting up, it may be necessary when the experiment requires operator interaction.

A. Requirements for configuration file

We should request following for general configurations. General configuration should describe:

- all the types of VMs and number of creating copies;
- the parameters for each VM type (e.g., allocated resources, path to disk image);
- virtualization parameters (type of hypervisor);
- network settings;
- post-install scripts.

This set of parameters is enough to creating wide class of research models.

VII. AUTHOR'S APPROACH

At present time, we have created an automatic system to deploy a VM using the libvirt library. It supports the deployment of the model of a distributed system based on a set of VM types. Figure 2 schematically shows the general scheme of polygon.

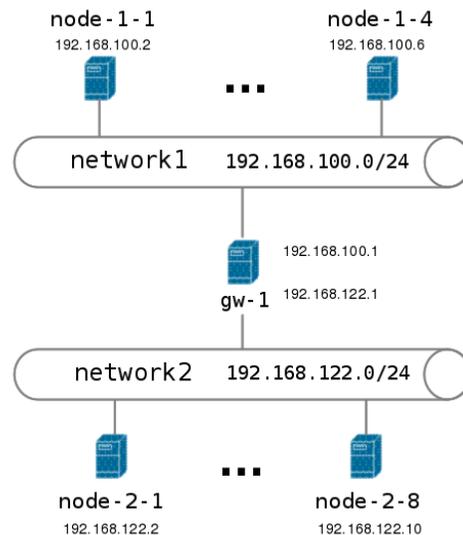


Fig. 2. System diagram

An algorithm includes following steps:

- creating a universal configuration in JSON (figure 3);
- preparing VM template disk images;
- incremental cloning of template images, network settings customization;
- creating XML-descriptions for each VM instance;
- creating a set of VMs based on these XML-descriptions via libvirt methods.

The first two steps are making manually by the operator, and the rest is automated.

```
{
  "type" : "kvm",
  "machines" : {
    "gw" : {
      "number" : 1,
      "memory" : "524288",
      "disk" : "/images/1.img",
      "network" : [network1, network2]
    },
    "node-1" : {
      "number" : 4,
      "memory" : "262144",
      "disk" : "/images/2.img",
      "network" : [network1]
    },
    "node-2" : {
      "number" : 8,
      "memory" : "262144",
      "disk" : "/images/3.img",
      "network" : [network2]
    }
  },
  "netconfig" : "Network.cfg"
}
```

Fig. 3. Example configuration of the test model

Figure 3 shows the configuration of the system consisting of three types of nodes: "gw" (1 item), "node-1" (4 items) and "node-2" (8 items). There specified the size of allocated memory and the path to disk image for each type of nodes. "Network" option contains a list of used virtual networks, which are described in the configuration file "Network.cfg" (figure 4).

"Network1" in this example – is the network connecting nodes of type "node-1", and "network2" connecting "node-2" nodes. "Gw-1" node plays the role of the router and has connected to both networks.

It should be noted that the operator sets the configuration manually, but the rest is automated. Deployed system have automatically configured remote access via SSH and the keys stores on the host machine.

A. Deployment on a distributed host system

There are also a possibility to use distributed host system for deployment. It means that you have a number of physical

```
{
  "networks" : {
    "network1" : {
      "range_start" : "192.168.100.2",
      "range_end" : "192.168.100.255",
      "gateway" : "192.168.100.1",
      "netmask" : "255.255.255.0",
      "nat" : "yes"
    },
    "network2" : {
      "range_start" : "192.168.122.2",
      "range_end" : "192.168.122.255",
      "gateway" : "192.168.122.1",
      "netmask" : "255.255.255.0",
      "nat" : "yes"
    }
  }
}
```

Fig. 4. Example configuration of the network (Network.cfg)

machines and the operator can deploy a large-scale distributed system based on several host machines. For example, if we have four hosts with 100 VMs, summary there are 400 VMs.

Distributed deployment system requires following:

- all hosts are connected to the VPN-network;
- there are one controlling host (Deployment Server);
- NFS-server is installed on the Deployment Server;
- configuration file "hosts.json" (figure 5) is stored on the DS contains IP-addresses of all hosts;
- DS have remote access to other hosts via SSH.

```
{
  "hosts" : {
    "Deployment Server" : {
      "address" : "10.8.0.1",
      "config" : "server_conf.json",
      "netconfig" : "server_netw.json"
    },
    "host1" : {
      "address" : "10.8.0.4",
      "config" : "host1_conf.json",
      "netconfig" : "host1_netw.json"
    },
    "host2" : {
      "address" : "10.8.0.8",
      "config" : "host2_conf.json",
      "netconfig" : "host2_netw.json"
    }
  }
}
```

Fig. 5. Example configuration of the distributed deployment system

To start deployment process the operator should launch server application on DS, and then launch client applications on hosts.

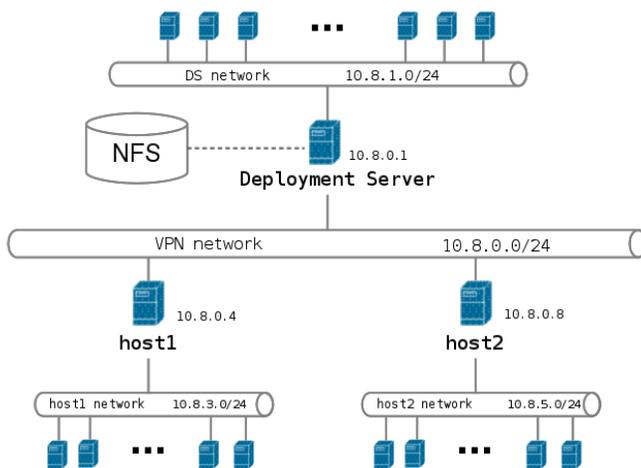


Fig. 6. Distributed system diagram

Figure 6 shows a distributed system with general NFS-store. Additionally, there are possibility to allow access to host's virtual networks to other hosts using VPN-tunnel.

VIII. EXPERIMENTS

Using the developed software was made a series of experiments on the parallel tasks execution. Experiments showed that deployed virtual model of the distributed computer system satisfies the requirements for the system functions. Particularly, the created nodes can execute remotely received tasks.

The test were conducted using remote access via SSH. There were created program scripts which launches DDoS-attack against one chosen node automatically, while the other nodes were attacking. Simulated attacks were successful. A network access to the victim VM was blocked.

The scenario of the experiment is parameterized, i.e. you can change parameters of the experiment such as number of nodes, addresses, launching tasks, but the used script is universal.

The experiments were performed with Intel i5-3450 based system with 16 GB RAM. There was deployed a model of distributed system consisting of 200 nodes on this host. Elapsed time of deployment was 24 minutes. Deduplication technologies such as UKSM led to this result. At the launch time the memory use was 14 GB, and one hour later it was reduced to 7 GB.

IX. CONCLUSION

As a result of this research we have created a software prototype for deploying a virtual model of a distributed computer system. Virtualization-based models of grid computing systems, produced with the help of the developed software,

were used to simulate various processes in such systems including their regular functioning and the reaction to denial-of-service attacks.

Further work is planned to add support for automated deployment of the software for distributed and parallel computing, such as the Globus Toolkit and MPI implementations. Beside that, we plan to add support for nested virtualization in order to create virtualization-based models for the systems which use virtualization technologies by themselves—cloud computing systems being a notable example.

REFERENCES

- [1] Grossman, R., et al. The open cloud testbed: A wide area testbed for cloud computing utilizing high performance network services. Preprint arXiv:0907.4810. 2009.
- [2] Krestis A., et al. Implementing and evaluating scheduling policies in gLite middleware // *Concurrency and Computations: Practice and Experience*. Wiley, 2012. URL: http://www.ceid.upatras.gr/faculty/manos/files/papers/cpe_2832_Rev_EV.pdf
- [3] Libvirt. The virtualization API. URL: <http://libvirt.org>
- [4] Chef. // Opscode. URL: <http://www.opscode.com/chef/>
- [5] Puppet Labs: IT Automation Software for System Administrators. // Puppet Labs. 2013. URL: <https://puppetlabs.com/>
- [6] Yong Kui Wang, Jie Li. Automate VM Deployment // IBM.COM. 2009. URL: <http://www.ibm.com/developerworks/linux/library/l-auto-deploy-vm/>
- [7] Vagrant // HashiCorp. 2013. URL: <http://www.vagrantup.com/>
- [8] VMware vSphere Auto Deploy: Server Provisioning // VMware. 2013. URL: <http://www.vmware.com/products/datacenter-virtualization/vsphere/auto-deploy.html>

Intelligent search based on ontological resources and graph models

Chugunov A.P.

Computer Science Department
Perm State National Research University
Perm, Russia
chugunov@permedu.ru

Lanin V.V.

Department of Business Informatics
National Research University Higher School of Economics
Perm, Russia
lanin@perm.ru

Abstract— This paper describes our approach to document search based on the ontological resources and graph models. The approach is applicable in local networks and local computers. It can be useful for ontology engineering specialists or search specialists.

Keywords—ontology; semantic; search; graph; document.

I. INTRODUCTION

Today the amount of electronic documents is very large and information searching remains to be a very hard problem. The majority of search algorithms, applicable in local networks, based on full-text search and don't take into account the semantics of a query or document. And good statistical methods can't be used in the local documents repository.

Mathematical and statistical (latent semantic search), graph (the set of documents presented as directed graph), ontological (the searching by existing ontologies) methods are used in computer search [1]. All of them have some imperfection [2].

In spite of this, the tandem of latent semantic and graph methods give very good results. The majority of internet search engines use it [3]. But graph method is not applicable in local networks or local computers [2]. And this approach doesn't let the consideration of semantic context of documents or all parts of search.

So, the task of semantic search hasn't been decided yet. And the newest search algorithms on the internet remain inapplicable in local networks or local computers.

If we combine the tandem with the third, semantic method, we get a possibility to decide the problem of taking into account a semantics. We have chosen ontologies as a semantic method because it allows solving the problem of a document directed graph building too. The building of full ontologies is not required.

The aim of our survey is to unite three different search approaches into one.

II. DESCRIPTION OF RELATED WORK

We observed the most popular algorithms of different search approaches:

1. Namestnikov's A. M. algorithm informational search in semantic project repository [4];
2. information search based on semantic metadescription [5];
3. In-Degree algorithm [6];
4. PageRank algorithm [6];
5. HITS algorithm [7].

The survey was made with a tendency towards on ontology applicable in approach, precision and recall of search results. The extract of survey [3] is presented in table 1.

TABLE I. THE SURVEY OF SEARCH ALGORITHMS.

	Using of ontologies	Ontology applicable	Precision	Recall
Namestnikov's A. M. algorithm informational search in semantic project repository	Yes	Yes	85%	69%
Information search based on semantic metadescription	Yes	Yes	97%	85%
In-Degree algorithm	No	Yes	75%	47%
PageRank algorithm	No	Yes	81%	66%
HITS algorithm	No	Yes	63%	78%

The highest result of precision made by information search based on semantic metadescription. But this algorithm requires a lot of ontology building, because it needs human participation. [2]

So, we decided to use HITS algorithm, because it has the best result and it's applicable to our work.

The using of ontologies in HITS algorithm is planned on the stage of primary documents set forming, which satisfy the query, as well as on the stage of G_δ forming and changing.

III. DEFINITION

We used the following definition of ontology [5] as basic: ontology is a triplet $O = \langle X, R, F \rangle$ where

X – not empty set of concepts of subject area;

R – finite set of relations between concepts;

F – finite set of interpretation functions, adjusted on concepts and/or relations of ontology;

We must mention the fact, that R and F can be empty. Ontology can contain instances of classes – the classes with preset properties.

In our work we will use the changed definition of ontology: ontology is a pair $O = \langle X, R \rangle$ with some constraints on the concepts set and relations set [2].

Document in our paper is a set of properties of a real document, subject, content and document ontology. Properties of real document are any data about it, which isn't presented in the content, including metadata.

$$D = \langle R, C, O \rangle$$

R – set of document properties. A set of properties can be described by metadata standard “Dublin core” [4];

C – content, i.e. entry of the document;

O – document ontology.

IV. ALGORITHM DESCRIBING

Proposed algorithm consists of 5 steps:

1. Building ontology O by existing documents.
2. The second step is to enter a query by the user, i.e. the determination of the set of primary concepts $\{C_i\}$, is interesting for the user.
3. The third step is the allocation of a documents set A_p , that contains all or some from $\{C_i\}$. Denote this A_s .
4. The fourth step is executing a range algorithm with input: $\{C_i\}$ as a user query, A_s as a primary document set, O as a directed document graph.
5. Output results to the user.

A. Building ontology

The step is preparatory. On this step we solve the task of automatic document ontology building, selected document properties from unstructured text on the natural language.

After document ontology building we determine “link to” type links between documents. It is advisable to combine this links into separate ontology. During this process not existing files can be included in the set. These links must be placed in

the set because it allows making search of documents, which doesn't exist in the repository.

After that we get 2 levels of ontologies:

1. Document ontologies. Define them $\{O_1, O_2, \dots, O_n\}$, where n is amount of documents.
2. Documents links ontology. Define it O_L .

In addition, the subject area ontology O_p can be made. This ontology doesn't depend on documents D, it contains only the knowledge about the subject area. Building of O_p can be automatic or manual. It's main, that if the amount of documents subject areas will be large, large amount of O_p can spoil the results. It leads to anomalies, conflicts, ambiguity between ontologies.

B. Entry a query by user

It's the first step in search. The aim of that is to determine a set of concepts $Q = \{C_i\}$, which are interesting for the user.

From the query we select keywords, concepts. Next, we extend the set due to subject ontologies, if it exists. This extend will contain synonyms, definitions and else.

Besides, the part of ontology O_p is being built on this step. The part contains a user query. Afterwards we will use it for calculating the weight of documents.

C. Allocation of a documents set

The goal of this step is building a primary documents set $D_F = \{D_i\}$, which satisfies the user query Q. The set is not final and can be changed on the next step.

Since the set is not final, we use latent semantic search on this step. We choose it because it gives high speed of search and relatively high precision of results.

The primary set can be calculated by the following formula:

$$D_F = \{D_i | X \cap Q \neq \emptyset, X \in O, O \subset D_i\}$$

In this set come documents, which keywords and concepts X in document ontology O (we define it O_i) are crossing with $\{C_i\}$ in the user query Q.

After that, we assign weight of each document in D_F . This weight reflects semantic distance to user query. This weight can be calculated by

$$w_i = \text{avg}(\text{sim}_{sem}(t_1, t_2) | t_1 \in O_i \cap t_2 \in Q),$$

where

$$\text{sim}_{sem}(t_1, t_2) = \begin{cases} |k| \cdot \frac{\text{sim}_{sem}(s_1, s_2) + \text{sim}_{sem}(o_1, o_2)}{2}, & \text{if } k > 0 \\ |k| \cdot \frac{\text{sim}_{sem}(s_1, o_2) + \text{sim}_{sem}(o_1, s_2)}{2}, & \text{else} \end{cases}$$

where $k = \text{sim}_{sem}(p_1, p_2)$ is value of distance between predicates, and t_1, t_2 are triplets. Triplet is a set of three $\langle X_1, P, X_2 \rangle$ where X_1 and X_2 are ontology concepts, P is predicate, relation between X_1 and X_2 .

User query Q and documents D_i have ontology view O_Q and O_i . Each ontology divides into triplets t_1 and t_2 , which can be intersected in an ontology. Next, we calculate semantic distance in pairs. Semantic distance between a user query and a document calculation as average of semantic distances of them triplets. It allows to take into account not absolute coincidences.

If we combine O_L and $\{w_i\}$, we get weighted directed graph $G=\langle V,E\rangle$, where V is a set of documents $\{D_i\}$, some of them has a weight – a number. If number is missing, the weight we let 0. Set E – a set of directed arcs, which present the links between documents. Arcs E haven't weights because it's impossible to determine power of link automatically with needed accuracy between documents today.

D. Executing range algorithm

Primary documents set D_F are extending by documents, which have links (in or out) with documents from D_F . In algorithm exists parameter d – amount of documents, which can be added by document from R_δ . In the set must be added d or fewer documents with maximal weights (semantic distance). It's important, that the weight of adding document must be bigger than w_{min} . This rule rises precision and recall of the results.

Documents ranging process base on vertex weights and amount of in- and out- arcs. It allows to get semantically closer documents in the results, even if they have small amount of arcs or haven't them at al.

So, the result of the algorithm is a set of pairs $D_R=\langle D_i,r_i\rangle$, where

D_i – found document

r_i – rang of the document.

E. Output results to the user

The set D_R can be output to the user as a traditional list of documents ordered by their weights or in graphical mode – as a document graph.

V. CONCLUSION

In this work we developed, offered and described our information and documents search approach, which combine 3 most widespread methods. We described it mathematically.

Now we have started the first realization of this approach. As a starting subject area we have chosen the science papers and publications, because these documents meet the standards of typography.

REFERENCES

1. Gasanov E. E. Information storage and search complexity theory, *Fundamentalnaya i prikladnaya matematika*, vol. 15 (2009), no. 3, pp. 49–73.
2. Никоненко А.А. Обзор баз знаний онтологического типа / Штучний інтелект, 2009, № 4. С. 208-219.
3. Signorini A. A survey of Ranking Algorithms, <http://homepage.divms.uiowa.edu/~asignori/phd/report/a-survey-of-ranking-algorithms.pdf>
4. Наместников А. М. Интеллектуальный сетевой архив электронных информационных ресурсов / А. М. Наместников, Н. В. Корунова, А. В. Чекина // Программные продукты и системы, 2007, № 4. С. 10-13.
5. Гладун А.Я. Онтологии в корпоративных системах / А.Я. Гладун, Ю.В. Рогущина // Корпоративные системы. – 2006. – № 1. – С. 41-47.
6. K. Bharat, Henzinger M. R. Improved algorithms for topic distillation in a hyperlinked environment// In Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '98). ACM, 1998, New York, USA, p. 104-111.
7. Kleinberg J. Authoritative sources in a hyperlinked environment, *Journal of ACM (JASM)*, №4, 1999, pp. 85-86.

Intelligent Service for Aggregation of Real Estate Market Offers

Lanin V., Nesterov R.

Department of Business Informatics
National Research University Higher School of Economics
Perm, Russia
lanin@perm.ru; mistika93@mail.ru

Osotova T.

Computer science department
Perm State National Research University
Perm, Russia
hvestya@gmail.com

Abstract – This article contains the implementation description of a real estate market offers aggregator service. Advertisement analysis is made with the aid of ontologies. A set of ontologies to describe specific websites can be extended, so the aggregator can be used for many diverse resources.

Keywords – intelligent service; real estate; ontology

I. INTRODUCTION

Real estate agents constantly analyze different information flows, so intellectual analysis of real estate market offers and monitoring services are required for their efficient work. Most of this information is semistructured and in this case conventional processing is time-consuming. Real estate information resources are topical Internet resources, newspapers and special databases.

Information aggregation and structuring tasks are increasingly timely. Apart from that, it is necessary to address information duplication and inconsistency search tasks. Semistructured information and its heterogeneous resources implies application of artificial intelligence means: text mining, Semantic Web technologies and multi-agent technologies.

Our solution is to develop intelligent service to accumulate information on real estate market offers from different resources in a single database.

II. REAL ESTATE MARKET OFFERS AGGREGATORS CLASSIFICATION

The term “Aggregator” is used to describe Internet resources and services accumulating existing real estate market offers. Database completeness, data timeliness and fidelity, search and filtering capabilities and access price are the main features of aggregators [3].

Existing resources can be classified in two ways: by database areal coverage and by the way of organizing customer relations. According to the first classification resources can be divided into two groups: global ones based on a well-known web portal platform (“Yandex.Nedvizhymost” [2]) and local ones related to the regional real estate business projects. According to the second classification resources can be divided into following groups: on-line bulletin boards, electronic

versions of free advertisements newspapers, multilisting systems, information portals and meta-aggregators.

One of the first to appear was on-line bulletin board. It is usually chargeless and topically organized database. Bulletin board is arranged as a website, where anyone can place an advertisement and visitors can read it. According to experts’ opinions, on-line bulletin boards, created simultaneously with the growth of a real estate market, establish themselves a lot more firmly. New bulletin boards projects do not approach a market because they require significant capital and financial inputs. Specialists call bulletin boards a “dirty” database, i.e. disorganized and almost unregulated. Nowadays boards prevent real estate market from proper functioning, because, generally speaking, bulletin boards creators are not interested in information structuring and quality enhancing as well as in information exchange cost reduction of real estate market participants.

Electronic versions of free advertisements newspapers are also one of the core information aggregators. For instance, they include “Iz ruk v ruki” website. According to experts’ opinions, the main advantage, that allows this kind of resources to take the lead in their market segments, is that it combines newspaper concept with its electronic version. That is why, non-Internet users can also be involved, so much larger market coverage will be provided.

Among real estate brokers the most popular and in-demand kind of resources is multilisting systems. The major difference between real estate market aggregators in Russia and in western countries that in latter ones portals are owned by non-governmental organizations. Multilisting is a basis used by all market participants. For example, National Association of Realtors in USA owns the world largest real estate information aggregator “Realtor.com”. At present Russia has no global portals that would aggregate information on all real estate market offers. Commercial portals created as business projects in different Russian regions occupy this niche market.

Real estate information portals or specific (customer-oriented) websites are the most widespread aggregators of real estate information on the Internet. They are the projects that can capture its audience by having a database and providing information uniqueness, convenient delivery, wide range of analytical services, specific positioning and target audience

choosing means. Experts say that these portals appeal to users because they offer more specific information: news, analysis and wider range of search filters. Services of real estate information portals are more convenient than ones of multi-purpose websites because such portals are designed especially for keeping real estate information.

Social networks also can be called information aggregators. It is a CRM-direction that implies a step when a customer interacts with an agent. Nonetheless, today the distance between website-aggregators and social networks is shortening from the point of view on common features and applied services. In western countries this technologies are long since popular and mainly because due to Web 2.0 technologies a website visitor is becoming an information co-author and increase its reliance among other society members.

Meta-aggregator is a system accumulating real estate offers from several resources. The examples are “Skaner Nedvizhymosti” (rent-scaner.ru), “Choister” (choister.ru) and BLDR (bldr.ru). These resources offer extra services like intellectual advertisement search placed only by an owner, not by a broker.

III. DESCRIPTION OF SERVICE IMPLEMENTATION

A. Service architecture

To address automatic population of a real estate items database in the context of project on creating a real estate agency automation system an intelligent service was implemented. It extracts information on real estate items from unstructured advertisements placed on different resources. The solution is based upon an ontological approach. The general architecture of the implemented service is shown in fig. 1.

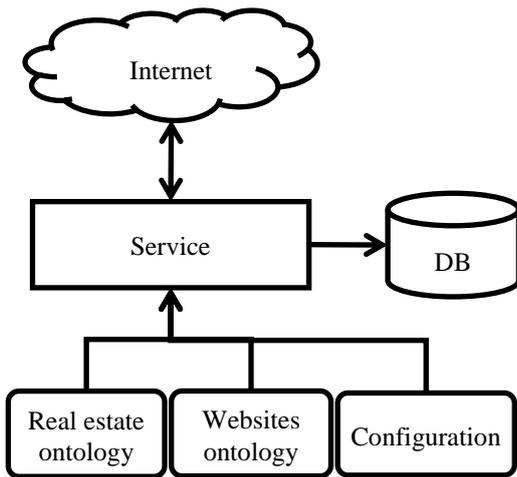


Fig. 1. Common service architecture

B. Service work layout

The general service work layout is shown in fig. 2.

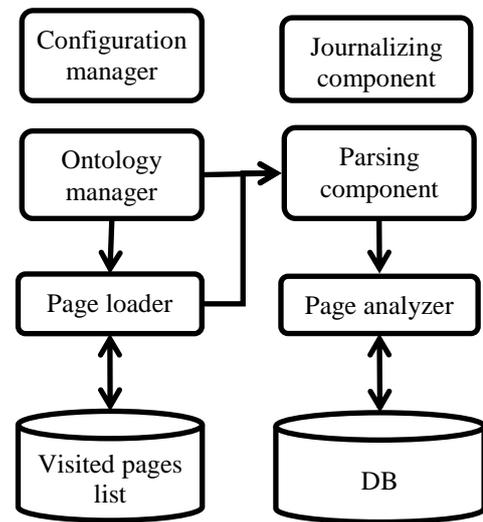


Fig. 2. Main modules of the service

Journalizing component makes a record of a service work (functioning). This record is used for service monitoring and debugging.

Configuration manager gives access to service settings and if necessary dynamically configures the service.

Ontology manager operates with ontological resources.

Page loader creates a local copy of a page and exercises its preprocessing. Information on visited pages is put into a special database. Due to this, during one loading session the loader will not visit the same pages; thus, it allows to improve service work. On basis of real estate websites ontology the loader extract information from the page. In this fashion page parser will have preprocessed text of a real estate advertisement, from which it extracts knowledge using real estate items ontology. Then this knowledge is unified (e.g. floor space can be converted to square meters).

Page analyzer makes an inference using real estate items ontology and captured knowledge as well as it checks several additional heuristics, after that it forms an object to be put into a correspond database.

C. Real estate websites ontology

Real estate websites ontology keeps specific websites settings. We are interested in keeping following parameters:

- 1) Position on a page, where the information will be found most likely and a description to have a title of this information;
- 2) Position on a page, where useful references can be found;
- 3) Description of filters to toss out “garbage” references for our service;
- 4) “Page turning” mechanism settings (more details on this are given below).

D. Real estate items ontology and regular expressions

Real estate items ontology keeps general domain concepts and their interconnections.

While parsing pages, the service attempts to “bind” specific concepts using ontology knowledge. Specific regular expressions are attached to each ontology concepts. There are two categories of regular expressions: general and website-adjusted. The latter can be used for binding only at specific websites and in general they are wrong (they allows to parse specific wordings used on a website more effectively). General regular expressions come into action in general cases. Firstly, binding of specific concepts is implemented using website-adjusted regular expressions and then in case of failure using general ones.

A regular expression consists of two components: ones to show that coincidence was found and ones to show erroneous binding. For example, “telephone” concept is binding (i.e. there is phone line), however advertisements often give a placer phone number. The second type of components used to identify a situation when it is said about different concept.

Apart from that, while extracting knowledge from text specific figures are also being bound: e.g. “Flat floor space” or “Focal person phone number”. The general structure of regular expressions is the same with the above, but additionally there are several logic parts used to convert figures to a single system. For example, if the price in advertisement were in rubles per Are, the service will convert it to rubles per square meter.

E. “Page turning” mechanism

While analyzing real estate items advertisements websites structure, it was identified that there are often lists containing advertisements references. A website has a plenty of advertisements and they are placed on different pages, that is why page crossing is implemented with navigation buttons.

We develop a “page turning” mechanism to exercise a sequential page crossing. Settings required for it are kept in real estate websites ontology and custom for each website.

It stands to mention that reference click-through, when a part of list is loaded with JavaScript, is a bit difficult to process. It was addressed by using special classes.

F. Service settings and load list

Service settings include parameters responsible for service functioning. There are following parameters:

- 1) Load list path with addresses that will be scanned by the service.
- 2) A path for saving loaded pages;
- 3) Time lapse, in that the service will resume work (service functioning can be stopped when it scanned all addresses in a load list);

4) “Websites scanning depth”, i.e. maximum path length to be scanned by the service;

5) Flag showing whether to go to third-party websites in case of the “in-depth” search.

G. “Page turning” mechanism

While analyzing real estate items advertisements websites structure, it was identified that there are often lists containing advertisements references. A website has a plenty of advertisements and they are placed on different pages, that is why page crossing is implemented with navigation buttons.

We develop a “page turning” mechanism to exercise a sequential page crossing. Settings required for it are kept in real estate websites ontology and custom for each website.

It stands to mentions that reference click-through, when a part of list is loaded with JavaScript, is slightly difficult to process. It was addressed by using special classes.

H. Programming and software tools

The service was developed using Microsoft Visual Studio 2010 and C# programming language. Ontology was developed with the aid of Protégé ontology editor. Also, HtmlAgilityPack (for html-pages parsing) and OwlDotNetApi (for reading ontologies from a file) libraries were used.

IV. BENCHMARKING

The service demonstrates rather high accuracy rates. Approximately 97 per cent of all advertisements are recognized in an adequate way. In 93 per cent of the time advertisement attributes are recognized precisely. Recognition accuracy can be improved with the aid of adjusting ontology to the specific representation of an advertisement. Besides, logging component includes analytical tools to find a reason for a fail correlation and to recommend on required ontology settings.

V. CONCLUSION

In this paper we described the architecture and peculiar implementation properties of the service aggregating the real estate market offers. At the moment the pilot system of the service is implemented. One of the core features of this service is that it can be adjusted to analysis of new resources without changing program code; configuration is only about ontology editing. Also, in the context of this project and on the basis of the information kept in the system, we intend to develop an expert system on selection and estimating of real estate items.

REFERENCES

- [1] Segaran T., Evans C., Taylor J. Programming the Semantic Web, O'Reilly Media, 2009.
- [2] Что такое Яндекс.Недвижимость <http://help.yandex.ru/realty/>
- [3] Недвижимость online: агрегаторы <http://media-office.ru/?go=2082914&pass=f79e9c77f077cf1d060a615834c3c2d1>

An Approach to the Selection of DSL Based on Corpus of Domain-Specific Documents

E. Elokhov, E. Uzunova, M. Valeev, A. Yugov, V. Lanin

Department of Business Informatics

National Research University Higher School of Economics

Perm, Russian Federation

eugene.yelokhov@gmail.com, palgonuri@gmail.com, mt.vallev.1992@gmail.com, yugovas@live.ru, lanin@perm.ru

Abstract. Today many problems that are dedicated to a particular problem domain can be solved using DSL. Thus to use DSL it must be created or it can be selected from existing ones. Creating a completely new DSL in most cases requires high financial and time costs. Selecting an appropriate existing DSL is an intensive task because such actions like walking through every DSL and deciding if current DSL can handle the problem are done manually. This problem appears because there are no DSL repository and no tools for matching suitable DSL with specific task. This paper observes an approach for implementing an automated detection of requirements for DSL (ontology-based structure) and automated DSL matching for specific task.

Keywords: *ontologies, conceptual search, domain-specific language, semantic similarity of words*

I. INTRODUCTION

Nowadays metamodeling and DSL-based technologies (DSL – Domain Specific Language) [16] are widely used in information system developing. DSL is created for solving some specific problem. Almost every arising problem is similar to the one that was solved before. In this case it means that a suitable DSL was already implemented or an implemented DSL does not fully meet the requirements. Therefore, you can either find a ready-to-use DSL or complete and configure a DSL implemented earlier. This requires less costs rather than developing a completely new DSL.

So, there are two steps to select one of already existing DSL:

1. Determine the requirements for DSL.
2. Find out how closely each of DSL meets this requirements.

Requirements are determined by analyzing domain-specific documents or problem statement. Then a requirements ontology based on that analysis is generated.

To match a concrete DSL with generated ontology some matching metrics and DSL description formats must be defined. In this work the MetaLanguage system [1] allowing

DSL creation will be used. The use of MetaLanguage system is justified by its noticeable features:

- 1) *the ability to work with most common DSL notations;*
- 2) *DSL convertation from one notation to another;*
- 3) *exporting dsls to external systems.*

In summary, the input data will be:

- corpus of domain-specific documents;
- set of DSL descriptions.

The target output is a list (ordered by correspondence to the generated ontology) of appropriate DSLs that can handle the problem.

This paper shows generating process of requirements ontology based on domain-specific documents and how a particular DSL meets given requirements.

II. RELATED WORKS

Nowadays there are some information systems that let you create text-based ontology models of documents or let you define correspondence of ontology models thereby transform one model onto another one. We found two web-resources that let you create ontologies: OwlExporter and OntoGrid.

The core idea of OwlExporter is to take the annotations generated by an NLP pipeline and provide for a simple means of establishing a mapping between NLP (Natural Language Processing) and domain annotations on one hand and the concepts and relations of an existing NLP and domain-specific ontology on the other hand. The former can then be automatically exported to the ontology in form of individuals and the latter as data type or object properties [7].

The resulting, populated ontology can then be used within any ontology-enabled tool for further querying, reasoning, visualization, or other processing.

OntoGrid is an instrumental system for automation of creating domain ontology using Grid-technologies and text analysis in natural language [12].

This system has bilingual linguistic processor for retrieving data from text in natural language. Worth D. derivational dictionary is used as a base for morphological analysis [4]. It contains more than 3.2 million word forms. The index-linking process consists of 200 rules. “Key dictionary” is determined by words allocation analysis in text. The developers came up with new approach of revealing super phrase unities that consist of specific lexical units. The building of semantic net is carried out this way: the text is analyzed using text analysis system, semantic Q-nets are used as formal description of text meaning [18]. The linguistic knowledge base of text analysis system is set of simple and complex word-groups of the domain. This base can be divided into simple-relation-realization base and critical-fragment-set, that let you determine which ontology elements are considered in this text. The next step is to create and develop the ontology in the context of GRID-net. A well-known OWL-standard is used to draw the ontology structure.

Also three information systems were found that fulfill a function of transformation [10].

ATLAS Transformation Language is a part of the architecture of managing ATLAS model [6]. ATL is the language that let you describe initial model transformation into destination model.

GReAT (Graph Rewriting And Transformation) is the language of model transformation description, which is based on triple graph transformation method [4]. This transformation represents the set of graph sorted re-record rules that are applied to the initial model and as a result create the destination model.

VIATRA is pattern-based transformation language for graph models managing which combines two methods: mathematic formal description (based on graph transformation rules for model description) and abstract finite state automaton (for control flow description) [5].

The program resources described before are key functions that determine an appropriate DSL matching. Unfortunately, a software system, which implements all this functions, was not found.

In addition, the idea used in applications intended to transform the ontology can be implemented to determine the measure of DSL correspondence to ontology requirements.

III. APPROACH DESCRIPTION

The suggested approach of the DSL selection process consists of six stages that can be described as a series of sequential operations which should be implemented (fig. 1).

Firstly, a corpus of documents is processed. As a result, the key words (concepts related to specific domain) are retrieved. Secondly, when re-viewing the document, the relations between concepts are built. These concepts and relations form a semantic network. The next step is to eliminate synonymy (to merge nodes containing synonymic concepts). In order to

achieve this, a linguistic ontology is used. After that, it is necessary to transform “contracted” semantic network into ontology model, using the graph coarsening algorithm with implementing linguistic ontologies. The next step is to qualify the ontology model by a specialist. This step includes concepts editing and relations marking semantically.

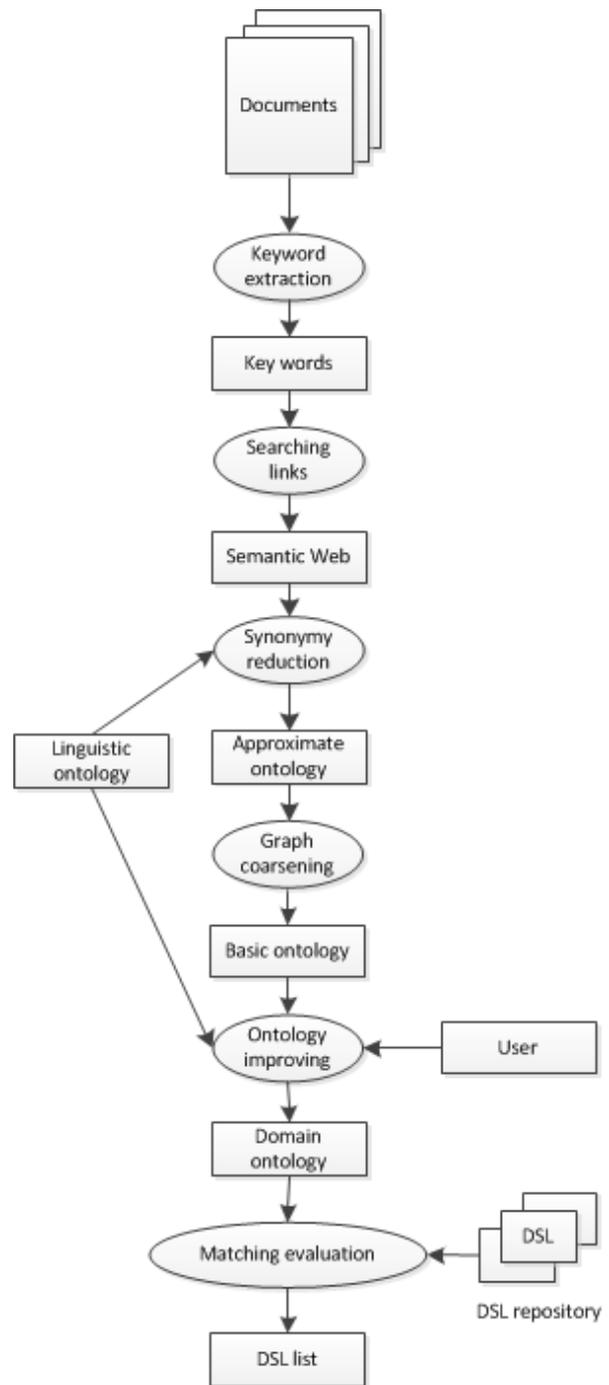


Figure 1. DSL selection process stages

When the ontology is complete, i.e. it meets user requirements, DSLs are taken from the repository, and the measures of DSLs correspondence to ontology requirements are calculated.

A. Keyword extraction

Using ontology is one of the most widespread ways to structure information on domain [11]. The formal ontology description is $O = \langle X, R, F \rangle$, where

- X – a finite set of domain terms,
- R – a finite set of relations between the terms,
- F – a finite set of interpretation functions.

Within the context of this paper, let us take a look at defining the set of terms and the set of relations.

Consider that basic terms in document are its key words-nouns. Researches related to finding key words in documents are based on frequency laws discovered by linguist and philosopher George Kingsley Zipf. The first law says that multiplication of word detection possibility and frequency rank is constant. The second law says that frequency and number of words with this frequency also have a relation.

Currently, for searching key words the pure Zipf's laws (TF-IDF) and also LSI (latent semantic indexing) algorithms are used. This research observes Zipf's laws, which are easily implemented, and a linguistic processing will be provided by program resources of Aot.ru.

As an example some university exam taking process is described. Consider that frequency analysis retrieved following keywords (fig. 2).

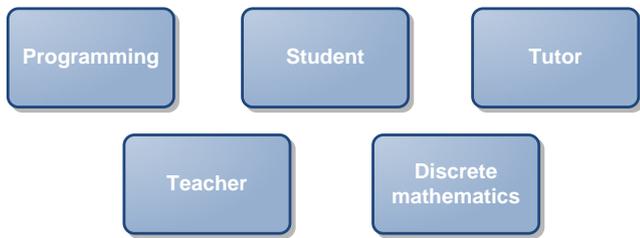


Figure 2. Exam taking keywords

B. Searching relations

As a result of frequency-response analysis we have a set of unlinked nodes (fig. 1). Now we have to define a set of relations, in other words to make disconnected graph a semantic net.

Semantic graph is weighted; its nodes are the terms of analyzed documents. The existence of edge between two nodes means that two terms are related semantically; weight of the edge is measure of semantic similarity [17].

Similarity measurement of ontology concepts can be calculated as follows:

1. Jaccard similarity coefficient [8]:

$$K_j = \frac{c}{a+b-c}$$

It's a statistic used for comparing the similarity and diversity of sample sets, where a – frequency of

occurrence of first term, b – frequency of occurrence of second term, c – frequency of occurrence of joint terms.

2. Mutual information [2]:

$$MI = \sum_{u=\{0,1\}} \sum_{v=\{0,1\}} P(u,v) \log_2 \frac{P(u,v)}{P(u)P(v)} \approx \sum_{u=\{0,1\}} \sum_{v=\{0,1\}} \frac{(u,v)}{N} \log_2 \frac{(u,v)}{(u)(v)} N$$

where u, v – terms retrieved from the document; (u) – frequency of occurrence of u , (v) – frequency of occurrence of v , (u, v) – frequency of occurrence of joint u and v .

Point mutual information may be calculated as [2]:

$$PMI(u,v) = p\left(\frac{(u,v)}{p(u)p(v)}\right).$$

After calculating measurements of ontology concepts they must be averaged [15]. Based on average measurement, keywords become connected. As a result the semantic net (fig. 3) is created.

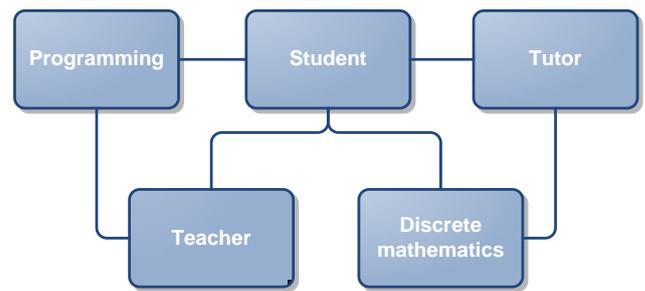


Figure 3. Exam taking semantic network

C. Synonymy reduction

Each concept is searched in linguistic ontology and those marked as synonyms are being contracted to a single node.

We are going to use *WordNet*, the semantic net, which was created at the Cognitive Science Laboratory of Princeton University. Its dictionary consists of four nets: nouns, verbs, adjectives and adverbs because they follow different grammatical rules. The basic dictionary unit is *synset*, combining words with similar meaning. It is also the node of the net. Synsets may have a few semantic relations like: *hypernym* (breakfast → eating), *hyponym* (eating → dinner), *has-member* (faculty → professor), *member-of* (pilot → crew team), *meronym* (table → foot), *antonym* (leader → follower). Different algorithms are widely used, for instance, the ones that take into account the distance between conceptual categories of words and hierarchical structure of *WordNet* ontology.

Linguistic ontology showed that example's *tutor* and *teacher* concepts are synonyms, so this concepts contract into one node (fig. 4).

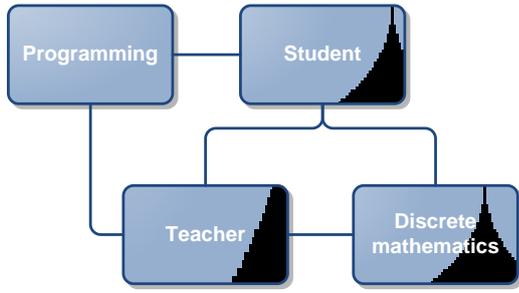


Figure 4. Exam taking semantic network after synonymy reduction

D. Graph coarsening

The next step is to transform the semantic net into ontology model. In general it's graph coarsening problem [5]. Classic methods of solving this problem are based on iterative contraction of adjacent nodes of graph G_α into nodes of graph $G_{\alpha+1}$, where $\alpha = 0, 1, 2, \dots$ – number of iteration, $G(0) = G(O)$. As a result the edge between two of graph G_α is removed and the multinode of graph $G_{\alpha+1}$ is created. [9].

When two nodes are replaced by one node (during the contraction), the values of these nodes are replaced by the value of parent node from linguistic ontology.

In example *programming* and *discrete mathematics* concepts are coarsened into one node (fig. 5).

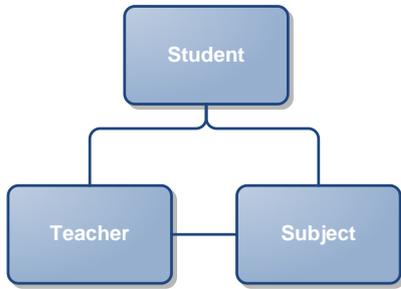


Figure 5. Exam taking semantic network after graph coarsening

E. Ontology improving

At this step we have a base ontology, representing criteria for DSL matching. However, it has some disadvantages:

- 1) no semantic relations representation;
- 2) unnecessary concepts may appear (this are useless for current task, but were generated during the analysis);
- 3) essential concepts could be missed during analysis.

To fix these disadvantages, this base ontology should be edited by human (specify relation semantics, add or delete concepts). Obviously, the more accurate will be ontology model, the more accurate DSL will be matched.

Consider that specialist renamed “*Subject*” to “*Exam*”, and removed relation between *student* concept and *teacher* concept, and added the semantic meanings to remaining

relations (student takes an exam and teacher grade an exam). The result is shown in fig. 6.

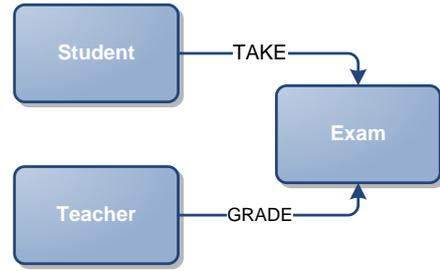


Figure 6. Exam taking ontology

F. Matching evaluation between DSL and created ontology

Comparison of ontologies comes down to calculation or relations revelation between the terms of two ontologies based on different lexical or structure methods. The result of this comparison represents a set of correspondences between the entities that are related semantically.

In order to assess how similar ontologies are, the extent of isomorphism should be measured.

Two graphs $(V1;E1; g1)$ and $(V2;E2; g2)$ are isomorphic if there are bijections:

$$f1 : V1 \rightarrow V2 \text{ and } f2 : E1 \rightarrow E2$$

so that for each edge

$$\begin{cases} a \in E1 \\ g1(a) = x - y \end{cases}$$

if and only if

$$g2[f2(a)] = f1(x) - f1(y).$$

It is not always easy to establish if two graphs are isomorphic or not. An exception is the case where the graphs are simple. In this case, we just need to check if there is a bijection

$$f: V1 \rightarrow V2,$$

which preserves adjacent vertices. If the graphs are not simple, we need more sophisticated methods to check for when two graphs are isomorphic

In our case, we should place emphasis that two graphs are not going to be isomorphic. However, the higher extent of isomorphism is, the more suitable current graph is.

The linguistic ontologies will have huge impact on the extent of isomorphism. For instance, if current node in the first graph was happened to describe a person and current node in the second graph described the document, isomorphism substitution would not exist in this context. At this moment, we are developing linguistic ontology-based algorithm for measuring how isomorphic two graphs are.

IV. CONCLUSION AND FUTURE WORK

In this paper a problem of matching a suitable DSL for specific task was observed.

The requirements for DSL are based on domain documents analysis. Requirements are formed as ontological model which is generated in two steps: defining concepts using frequency analysis of terms found and defining relations based on average weighted score obtained using Jaccard index and mutual information index.

The second step of DSL matching is comparison of DSL's that was implemented earlier with ontology based on domain documents analysis. The core of this comparison is the method of determining graphs' isomorphism and semantic match is controlled by linguistic ontology.

The further work is devoted to increasing the number of methods used to create more relations in the ontology model. This will improve the accuracy of average weighted score of concept relationship. Furthermore the DSL comparison on different levels will be observed (hierarchical structure comparison).

REFERENCES

- [1] A.O. Sukhov, L.N. Lyadova "MetaLanguage: a Tool for Creating Visual Domain-Specific Modeling Languages", Proceedings of the 6th Spring/Summer Young Researchers' Colloquium on Software Engineering, SYRCoSE 2012, Пермь: Институт системного программирования Российской академии наук, 2012, pp. 42-53
- [2] Centre for the Analysis of Time Series website. [Online]. Available: <http://cats.lse.ac.uk/homepages/liam/st418/mutual-information.pdf>
- [3] D. Balasubramanian "The Graph Rewriting and Transformation Language: GREAT". [Online]. Available: http://www.isis.vanderbilt.edu/sites/default/files/great_easst.pdf
- [4] D. Worth, A. Kozak, D. Johnson "Russian Derivational Dictionary", New York, NY: American Elsevier Publishing Company Inc, 1970
- [5] G. Karypis, V. Kumar "Multilevel k-way Partitioning Scheme for Irregular Graphs", Journal of Parallel and Distributed Computing, 96-129, 1998
- [6] J. Bezivin "An Introduction to the ATLAS Model Management Architecture". [Online]. Available: <http://www.ie.inf.uc3m.es/grupo/docencia/reglada/ASDM/Bezivin05b.pdf>
- [7] R. Witte, N. Khamis, and J. Rilling, "Flexible Ontology Population from Text: The OwlExporter" Dept. of Comp. Science and Software Eng. Concordia University, Montreal, Canada. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2010/pdf/932_Paper.pdf
- [8] R. Real, J. Vargas, "The Probabilistic Basis of Jaccard's Index of Similarity" [Online]. Available: <http://sysbio.oxfordjournals.org/content/45/3/380.full.pdf>
- [9] А. Карпенко "Оценка релевантности документов онтологической базы знаний". [Online]. Available: <http://technomag.edu.ru/doc/157379.html>
- [10] А. Сухов "Методы трансформации визуальных моделей". [Online]. Available: <http://www.hse.ru/pubs/share/direct/document/68390345>
- [11] В. Аверченков, П. Казаков "Управление информацией о предметной области на основе онтологий". [Online]. Available: <http://www.pandia.ru/text/77/367/22425.php>
- [12] В. Гусев "Механизмы обнаружения структурных закономерностей в символических последовательностях", 47-66, 1983
- [13] В. Гусев, Н. Саломатина "Алгоритм выявления устойчивых словосочетаний с учётом их вариативности (морфологической и

- комбинаторной". [Online]. Available: <http://www.dialog-21.ru/Archive/2004/Salomatina.htm>
- [14] Г. Белоногов, И. Быстров, А. Новоселов и другие "Автоматический концептуальный анализ текстов" НТИ, сер. 2, № 10, с. 26-32, 2002
- [15] И. Мисуно, Д. Рачковский, С. Слипченко "Векторные и распределенные представления, отражающие меру семантической связи слов". [Online]. Available: http://www.immsp.kiev.ua/publications/articles/2005/2005_3/Misuno_03_2005.pdf
- [16] Л. Лядова "Многоуровневые модели и языки DSL как основа создания интеллектуальных CASE-систем". [Online]. Available: http://www.hse.ru/data/2010/03/30/1217475675/Lyadova_LN_2.pdf
- [17] М. Гринева, М. Гринев, Д. Лизоркин "Анализ текстовых документов для извлечения тематически сгруппированных ключевых терминов". [Online]. Available: http://citforum.ru/database/articles/kw_extraction/2.shtml#3.3
- [18] Н. Загоруйко, А. Налётов, А. Соколова и другие "Формирование базы лексических функций и других отношений для онтологии предметной области". [Online]. Available: <http://www.dialog-21.ru/Archive/2004/Zagorujko.htm> M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

Beholder Framework

A Unified Real-Time Graphics API

Daniil Rodin

Institute of Mathematics and Computer Science
Ural Federal University
Yekaterinburg, Russia

Abstract—This paper describes Beholder Framework, which is a set of libraries designed to provide a single low-level API for modern real-time graphics that combines clarity of Direct3D 11 and portability of OpenGL. The first part of the paper describes the architecture of the framework and its reasoning from the point of view of developing a cross-platform graphics application. The second part describes how the framework overcomes some most notable pitfalls of supporting both Direct3D and OpenGL that are caused by differences in design and object models of the two APIs.

Keywords—*real-time graphics; API; cross-platform; shaders; Direct3D; OpenGL;*

I. INTRODUCTION

Real-time graphics performance is achieved by utilizing hardware capabilities of a GPU, and to access those capabilities there exist two “competing” API families, namely Direct3D and OpenGL. While OpenGL is the only option available outside Windows platform, it has some significant drawbacks when compared to Direct3D including overcomplicated API [1] and worse driver performance [2]. For this reason, developers, who are working on a cross-platform graphics application that must also be competitive on Windows, have to support both Direct3D and OpenGL, which is a tedious and time-consuming work with many pitfalls that arise from the design differences of Direct3D and OpenGL.

This paper describes a framework that solves those issues by providing an API that is similar to Direct3D 11, while being able to use both Direct3D and OpenGL as back-ends. The former allows developers to use well-known and well-designed programming interfaces without the need to learn completely new ones. The later allows applications developed using the framework to be portable across many platforms, while maintaining the Direct3D level of driver support on Windows.

II. WHY OPENGL IS NOT SUFFICIENT

Since OpenGL provides a real-time graphics API that is capable of running on different platforms, including Windows, it might look like an obvious API of choice for cross-platform development. But if you look at the list of best-selling games [3] of the last ten years (2003 – 2012), you will notice that almost every game that has a PC version [3] uses Direct3D as

main graphics API for Windows and most of them are Direct3D (and thus, Windows) – only.

If OpenGL was at least near to being as powerful, stable, and easy to use as Direct3D, it would be irrational for developers to use Direct3D at all. Especially so for products that are developed for multiple platforms, and thus, already have OpenGL implementations.

These two facts bring us to a conclusion that, in comparison to Direct3D, OpenGL has some significant drawbacks.

In summary, those drawbacks can be divided into three groups.

The first reason of Direct3D dominance is that from the Direct3D version 7 and up, OpenGL was behind in terms of major features. For example, GPU-memory vertex buffers, that are critical for hardware T&L (transform and lighting), appeared in OpenGL after almost four years of being a part of Direct3D, and it took the same amount of time to introduce programmable shaders as a part of the standard after their appearance in Direct3D 8. [4]

And even today, when the difference between Direct3D 11 and OpenGL 4.3 features is not that noticeable, some widely used platforms and hardware do not support many important of them for OpenGL. For example, OS X still only supports OpenGL up to version 3.2. Another example is Intel graphics hardware that is also limited to OpenGL 3.x, and even OpenGL 3.x implementation has some major unfixed bugs. For instance, Intel HD3000 with current drivers does not correctly support updating a uniform buffer more than once a frame, which is important for efficient use of uniform buffers (a core OpenGL feature since version 3.1).

The third OpenGL drawback is very subjective, but still important. While trying to achieve backwards-compatibility, Khronos Group (an organization behind OpenGL) was developing OpenGL API by reusing old functions when possible, at the cost of intelligibility (e.g. `glBindBuffer`, `glTexImage3D`). This resulted in an overcomplicated API that does not correspond well to even the terms that the documentation is written in and still suffers from things like bind-to-edit principle [1]. On the other hand, Direct3D is being redesigned every major release to exactly match its capabilities, which makes it significantly easier to use.

III. ALTERNATIVE SOLUTIONS

Beholder Framework is not the first solution for the problem of combining Direct3D and OpenGL. In this section we will discuss some notable existing tools that provide such abstraction, and what are their differences from the Beholder Framework.

A. OGRE

OGRE (Open Graphics Rendering Engine) [5] is a set of C++ libraries that allow real-time rendering by describing a visual scene graph that consists of camera, lights, and entities with materials to draw, which are much higher-level terms than what APIs like Direct3D and OpenGL provide.

While this was the most natural way of rendering in the times of fixed-function pipeline, and thus, providing this functionality in the engine was only a plus, nowadays rendering systems that are not based on scene graph are becoming more widespread because the approach to performance optimizations has changed much since then [6, 7]. The other aspect of OGRE API being higher-level than Direct3D and OpenGL is that it takes noticeably longer to integrate new GPU features into it, since they must be well integrated into a higher-level object model that was not designed with those capabilities in mind.

Therefore, even though OGRE is providing rendering support for both Direct3D and OpenGL, it is not suited for those applications that require different object model or newer GPU features.

B. Unity

Unity [8] is a cross-platform IDE and a game engine that allows very fast creation of simple games that expose some predefined advanced graphics techniques like lightmaps, complex particle systems, and dynamic shadows.

Unity provides excellent tools for what it is designed for, but has even less flexibility than OGRE in allowing implementation of non-predefined techniques. It also forces an IDE on the developer, which, while being superb for small projects, is in many cases unacceptable for larger ones.

C. Unigine

Unigine [8] is a commercial game engine that supports many platforms, including Windows, OS X, Linux, PlayStation 3, and others while using many advanced technologies and utilizing low-level API to its limits. Having said that, Unigine is still an engine which forces the developer to utilize the graphics the specific way instead of providing a freedom like low-level APIs do.

In comparison to all the discussed solutions, Beholder Framework aims to provide the freedom of a low-level API (namely, Direct3D 11) while maintaining portability of supporting both Direct3D and OpenGL.

IV. BEHOLDER FRAMEWORK ARCHITECTURE

Beholder Framework is designed as a set of interfaces that resemble Direct3D 11 API, and an extensible list of implementations of those interfaces. The framework is being developed as a set of .NET libraries using the C# language, but it is designed in such a way that porting it to C/C++ will not pose any significant problems if there will be a demand for that.

All the interfaces and helper classes are stored in the Beholder.dll .NET assembly including the main interface – `Beholder.IEye` that is used to access all the implementation-specific framework capabilities. In order to get an implementation of this interface, one can, for example, load it dynamically from another assembly. This is a preferred way since it allows using any framework implementation without recompiling an application. At the time of writing, there are three implementations of the framework – for Direct3D 9, Direct3D 11, and OpenGL 3.x/4.x.

When the instance of the `Beholder.IEye` is acquired, one can use it to perform four kinds of tasks that are important for initializing a graphics application. The first one is enumerating graphics adapters available for the given system along with their capabilities. By doing this, one can decide what pixel formats to use, what display modes to ask the user to choose from, and whether some specific features are available or not. The second task is creating windows or preparing existing ones for drawing and capturing user input. The third one is initializing a graphics device, which is a main graphics object that holds all the graphics resources and contexts (corresponds to the `ID3D11Device` interface of Direct3D 11). Finally, the fourth task that `Beholder.IEye` can be used for is initializing a “game loop” – a specific kind of an infinite loop that allows the application to interact with the OS normally.

Another useful feature that the framework provides at the `Beholder.IEye` level is a validation layer. It is an implementation of the interfaces that works like a proxy to a real implementation while running a heavy validation on the interface usage. This is useful for debugging purposes, and since it is optional, it will not affect performance of a release build.

When the device is initialized and the game loop is running, an application can use Beholder Framework in almost the same way it could use Direct3D with only minor differences. The only exception to this is a shader language.

V. UNIFYING SHADERS

Even though both Direct3D 11 and OpenGL 4.3 have similar graphics pipelines, and thus, same types of shaders, they provide different languages to write them, namely HLSL and GLSL respectively. Compare, for example, these versions of a simple vertex shader in two languages:

A. HLSL

```
cbuffer Transform : register(b0)
{
    float4x4 World;
    float4x4 WorldInverseTranspose;
};
```

```

cbuffer CameraVertex : register(b1)
{
    float4x4 ViewProjection;
};

struct VS_Input
{
    float3 Position : Position;
    float3 Normal : Normal;
    float2 TexCoord : TexCoord;
};

struct VS_Output
{
    float4 Position : SV_Position;
    float3 WorldPosition : WorldPosition;
    float3 WorldNormal : WorldNormal;
    float2 TexCoord : TexCoord;
};

VS_Output main(VS_Input input)
{
    VS_Output output;
    float4 worldPosition4 = mul(float4(input.Position, 1.0), World);
    output.Position = mul(worldPosition4, ViewProjection);
    output.WorldPosition = worldPosition4.xyz;
    output.WorldNormal = normalize(
        mul(float4(input.Normal, 0.0), WorldInverseTranspose).xyz);
    output.TexCoord = input.TexCoord;
    return bs_output;
}

```

B. GLSL

```

#version 150

layout(binding = 0, std140) uniform Transform
{
    mat4x4 World;
    mat4x4 WorldInverseTranspose;
};

layout(binding = 1, std140) uniform CameraVertex
{
    mat4x4 ViewProjection;
};

in vec3 inPosition;
in vec3 inNormal;
in vec2 inTexCoord;

out vec3 outWorldPosition;
out vec3 outWorldNormal;
out vec2 outTexCoord;

void main()
{
    vec4 worldPosition4 = vec4(inPosition, 1.0) * World;
    gl_Position = worldPosition4 * ViewProjection;
    outWorldPosition = worldPosition4.xyz;
    outWorldNormal = normalize(
        (vec4(inNormal, 0.0) * WorldInverseTranspose).xyz);
    outTexCoord = inTexCoord;
}

```

As you can see, even though the shader is the same, the syntax is very different. Some notable differences are: many cases of different naming of same keywords (e.g. types), different operator and intrinsic function sets (e.g. while GLSL uses ‘*’ operator for matrix multiplication, in HLSL ‘*’ means per-component multiplication, and for matrix multiplication `mul` function is used instead), different input/output declaration approaches, and many others. Also notice how in HLSL output position is a regular output variable with a special HLSL semantic `SV_Position` (‘SV’ stands for ‘Special Value’), while in GLSL a built-in `gl_Position` variable is used instead.

To enable writing shaders for both APIs simultaneously, one would naturally want to introduce a language (maybe similar to one of the existing ones) that will be parsed and then translated to the API-specific language. And as you will see, Beholder Framework does that for uniform buffers, input/output, and special parameters (e.g. tessellation type). But because fully parsing and analyzing C-like code requires too

much time-commitment, the author decided to take a slightly easier approach for the current version of the framework.

Here is the same shader written in the ‘Beholder Shader Language’:

```

%meta
Name = DiffuseSpecularVS
ProfileDX9 = vs_2_0
ProfileDX10 = vs_4_0
ProfileGL3 = 150

%ubuffers
ubuffer Transform : slot = 0, slotGL3 = 0, slotDX9 = c0
    float4x4 World
    float4x4 WorldInverseTranspose
ubuffer CameraVertex : slot = 1, slotGL3 = 1, slotDX9 = c8
    float4x4 ViewProjection

%input
float3 Position : SDX9 = POSITION, SDX10 = %name, SGL3 = %name
float3 Normal : SDX9 = NORMAL, SDX10 = %name, SGL3 = %name
float2 TexCoord : SDX9 = TEXCOORD, SDX10 = %name, SGL3 = %name

%output
float4 Position: SDX9=POSITION0, SDX10=SV_Position, SGL3=gl_Position
float3 WorldNormal : SDX9 = TEXCOORD0, SDX10 = %name, SGL3 = %name
float3 WorldPosition : SDX9 = TEXCOORD1, SDX10 = %name, SGL3 = %name
float2 TexCoord : SDX9 = TEXCOORD2, SDX10 = %name, SGL3 = %name

%code_main
float4 worldPosition4 = mul(float4(INPUT(Position), 1.0), World);
OUTPUT(Position) = mul(worldPosition4, ViewProjection);
OUTPUT(WorldPosition) = worldPosition4.xyz;
OUTPUT(WorldNormal) = normalize(
    mul(float4(INPUT(Normal), 0.0), WorldInverseTranspose).xyz);
OUTPUT(TexCoord) = INPUT(TexCoord);

#define INPUT(x) bs_to_vertex_##x
in float3 bs_to_vertex_Position;
in float3 bs_to_vertex_Normal;
in float2 bs_to_vertex_TexCoord;

#define OUTPUT(x) bs_to_pixel_##x
#define bs_to_pixel_Position gl_Position
out float3 bs_to_pixel_WorldPosition;
out float3 bs_to_pixel_WorldNormal;
out float2 bs_to_pixel_TexCoord;

```

As you can see, `%meta`, `%ubuffers`, `%input`, and `%output` blocks can be easily parsed using a finite-state automaton and translated into either HLSL or GLSL in an obvious way (`slotDX9` and `SDX9` are needed for `vs_2_0` HLSL profile used by Direct3D 9). But to translate the code inside the main function, the author had to use a more ‘exotic’ tool – C macros, which, fortunately, are supported by both HLSL and GLSL.

Using macros helps to level out many of the language differences. Type names are translated easily, so are many intrinsic functions. Input and output macros for GLSL while being not so obvious are, nevertheless, absolutely possible. For example, input/output declaration that is generated by the framework for OpenGL looks simply like this.

While using macros does not make the unified shader language as beautiful and concise as it could be if it was being parsed and analyzed completely, it still makes writing a shader for all APIs at once not much harder than writing a single shader for a specific API, which is the main goal of a unified shader language.

VI. PITFALLS OF USING OPENGL AS DIRECT3D

Since Direct3D and OpenGL are being developed independently and only the fact that they must work with the same hardware makes them be based on similar concepts, it comes with no surprise that the APIs have many subtle differences that complicate the process of making one API

work like another. In this section we will discuss the most notable of such differences and ways to overcome them.

A. Rendering to a Swap Chain

While Direct3D, being tightly integrated into Windows infrastructure, applies the same restrictions for both on-screen (swap chain) render targets and off-screen ones, in OpenGL the restriction can be unexpectedly different. For example, at the time of writing, Intel HD3000 on Windows does not support multisampling and several depth-stencil formats for on-screen rendering that it supports for off-screen rendering using OpenGL.

To counter this, Beholder Framework uses a special off-screen render target and an off-screen depth-stencil surface when a developer wants to render to a swap chain, and then copies render target contents to the screen when `Present` method of a swap chain is called. This may seem like overkill, but as you will see in the next section, it has more benefits to it than just being an easy way to overcome OpenGL limitations.

B. Coordinate Systems

Despite the common statement that “Direct3D uses row vectors with left hand world coordinates while OpenGL uses column vectors with right hand world coordinates”, it is simply not true. When using shaders, an API itself does not even use the concept of world coordinates, and, as demonstrated in the previous section, GLSL has the same capabilities of working with row vectors (which means doing vector-matrix multiplication instead of a matrix-vector one) as HLSL. Nevertheless, there are still two notable differences between OpenGL and Direct3D pipelines that are related to coordinate systems.

The first difference is Z range of homogeneous clip space. While Direct3D rasterizer clips vertex with position p when $p.z / p.w$ is outside of $[0,1]$ range, for OpenGL this range is $[-1,1]$. Usually, for cross-platform applications it is recommended to use different projection matrices for Direct3D and OpenGL to overcome this issue [10]. But since we are controlling the shader code, this problem can be solved in a much more elegant way by simply appending the following code to the last OpenGL shader before rasterization:

```
gl_Position.z = 2.0 * gl_Position.z - gl_Position.w;
```

This way, Z coordinate of a vertex will be in the correct range, and since the Z coordinate is not used for anything else other than clipping at the rasterization stage, this will make Direct3D and OpenGL behave the same way.

The second coordinate-related difference is texture coordinate orientation. Direct3D considers the Y coordinate of a texture to be directed top-down, while OpenGL considers it to be directed bottom-up.

While the natural workaround for this difference would seem to be modifying all the texture-access code in GLSL shaders, such modification will significantly affect performance of shaders that do many texture-related operations. But since the problem lies in texture coordinates,

which are used to access the texture data, it can be also solved by inverting the data itself.

For texture data that comes from CPU side this is actually as easy as feeding OpenGL the same data that is being fed to Direct3D. Since OpenGL expects the data in bottom-up order, it can be inverted by feeding in in top-down order, in which it is expected by Direct3D.

For texture data that is generated on the GPU using render-to-texture mechanisms the easiest way to invert the resulting texture is just to invert the scene before rasterization by appending the following code to the last shader before the rasterization stage (the same place where we appended the Z-adjusting code):

```
gl_Position.y = -gl_Position.y;
```

This will make off-screen rendering work properly, but when rendering a final image to the swap chain, it will appear upside-down. But, as you can remember, we are actually using a special off-screen render target for swap-chain drawing. And thus, to solve this problem, we only need to invert the image when copying it to the screen.

C. Vertex Array Objects and Framebuffer Objects

Starting from version 3.0, OpenGL uses what is called “Vertex Array Objects” (usually called VAOs) to store vertex attribute mappings. This makes them seem to be equivalent to Direct3D Input Layout objects and makes one want to use VAOs the same way. Unfortunately, VAOs do not only contain vertex attribute mappings, but also the exact vertex buffers that will be used. That means that for them to be used as encapsulated vertex attribute mapping, there must be a separate VAO for each combination of vertex layout, vertex buffers, and vertex shader. Since, compared to just layout-shader combinations, such combination will most likely be different for almost every draw call in a frame, there will be no benefit from using different VAOs at all. Therefore, Beholder Framework uses a single VAO that is being partially modified on every draw call where necessary.

Unlike Direct3D 11 that uses “Render Target Views” and “Depth-Stencil Views” upon usual textures to enable render-to-texture functionality, OpenGL uses a special type of objects called “Framebuffer Objects” (usually called FBOs). When actually doing rendering to a texture, FBO can simply be used like a part of the device context that contains current render target and depth-stencil surface. But clearing render targets and depth-stencil surfaces, which in Direct3D is done using a simple functions `ClearRenderTargetView` and `ClearDepthStencilView`, in OpenGL also requires an FBO. Furthermore, this FBO must be “complete”, which means that render target and depth-stencil surface currently attached to it must be compatible.

When clearing a render target, this compatibility can be easily achieved by simply detaching depth-stencil from the FBO. But when clearing a depth-stencil surface, there must be a render target attached with dimensions not less than ones of the depth-stencil surface.

Therefore, to implement Direct3D 11 – like interface for render-to-texture functionality on OpenGL while minimizing the number of OpenGL API calls, Beholder Framework uses three separate FBOs for drawing, clearing render targets, and clearing depth-stencil surfaces. Render target FBO has depth-stencil always detached, and depth-stencil FBO uses a dummy renderbuffer object that is large enough for the depth-stencil surface being cleared.

VII. CONCLUSION AND FUTURE WORK

Supporting both Direct3D and OpenGL at the lowest level possible is not an easy task, but, as described in this paper, a plausible one. At the moment of writing a large part of Direct3D 11 API is implemented for Direct3D 9, Direct3D 11, and OpenGL 3.x/4.x and the project's source code is available on GitHub [11].

After collecting public opinion on the project, the author plans to implement the missing parts that include staging resources, stream output (transform feedback), compute shaders, and queries. After that the priorities will be a better shader language and more out-of-the-box utility like text rendering using sprite fonts.

REFERENCES

- [1] About 'bind-to-edit' issues of OpenGL API. <http://www.g-truc.net/post-0279.html#menu>
- [2] Performance comparison of Direct3D and OpenGL using Unigine benchmarks. <http://www.g-truc.net/post-0547.html>
- [3] List of best-selling PC video games. http://en.wikipedia.org/wiki/List_of_best-selling_PC_video_games
- [4] History of competition between OpenGL and Direct3D. <http://programmers.stackexchange.com/questions/60544/why-do-game-developers-prefer-windows/88055#88055>
- [5] Official site of the OGRE project. <http://www.ogre3d.org/>
- [6] "Scenegraphs: Past, Present, and Future". <http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future/>
- [7] Noel Llopis. "High-Performance Programming with Data-Oriented Design" Game Engine Gems 2. Edited by Eric Lengyel. A K Peters Ltd. Natick, Massachusetts 2011.
- [8] Official site of Unity project. <http://unity3d.com/>
- [9] Official site of Unigine project. <http://unigine.com/>
- [10] Wojciech Sterna. "Porting Code between Direct3D9 and OpenGL 2.0" GPU Pro. Edited by Wolfgang Engel. A K Peters Ltd. Natick, Massachusetts 2010.
- [11] Beholder Framework repository on GitHub. <https://github.com/Zulkir/Beholder>

Image key points detection and matching

Mikhail V. Medvedev

Technical Cybernetics and Computer Science Department
Kazan National Research Technical University
Kazan, Russia
mmedv@mail.ru

Mikhail P. Shleymovich

Technical Cybernetics and Computer Science Department
Kazan National Research Technical University
Kazan, Russia
shlch@mail.ru

Abstract—In this article existing key points detection and matching methods are observed. The new wavelet transformation based key point detection algorithm is proposed and the descriptor creation is implemented.

Keywords—key points, descriptors, SIFT, SURF, wavelet transform.

I. INTRODUCTION

Nowadays the information technology based on artificial intelligence develops rapidly. Typically database with sample based retrieval becomes the major component of such intelligent systems. Biometrical identification systems, image databases, video monitoring systems, geoinformation systems, video tracking and many other systems can be considered as an example of intelligent systems with such databases.

For intelligent systems database retrieval the sample of data is defined, major characteristics are extracted and then the objects with similar characteristics are found in the database. In many cases images become the database objects. So we need some mechanism of characteristics extraction and their following comparison for finding identical or similar objects.

At the same time the intelligent systems of object 3D reconstruction are widely spread. Such systems can be used in robotic technology, architecture, tourism and other spheres. There are two major approaches to the 3D reconstruction problem solving: active and passive methods. In active methods depth sensors are used. They should be attached to the object directly, but in many cases this is impossible because of inaccessibility of an object. Such systems become very complex and demand additional equipment.

In passive method case photo camera is used as the sensor. Camera gets photos of an object for different points of views. It is not necessary to use depth sensors in this approach, and that's why it can be applied in any cases under all conditions. However, the object reconstruction accuracy substantively depends on the quality of collected images and the reconstruction algorithm. The first step of such an algorithm is to compare the images and identify the same key points for the further 3D reconstruction scheme evaluation. For solving such problems we need a computationally simple mechanism for image comparison and their similarity finding. The key point based description of an object is not very complex and rather reliable, that's why it can be used in object identifying tasks.

So we can see that the problem of identifying the same object in different pictures becomes very actual.

Key points or salient points concern the major information about the image. They can be found in the areas, where the brightness of the image pixels significantly changes. The human eye finds such points in the image automatically. These points can be characterized with two major features: the amount of key points mustn't be very big; their location mustn't change accord to the changing of the image size and image orientation; key point position must not depend on the illumination. In this paper we discuss the most popular SIFT and SURF method, and also present the new method based on wavelet transformation.

II. EXISTING KEY POINT DETECTION METHODS

A. Harris Corner Detector

Harris corner detector [2] uses corners as the key points, because they are unique in two dimensions of the image and provide locally unique gradient patterns. They can be used on the image, when we have a small movement. The corner detection method looks at an image patch around an area centered at (x,y) and shifts it around by (u,v) . The method uses the gradients around this patch. The algorithm can be described in the following steps.

1. The calculation of the weighted sum of square difference between the original patch and the translated patch.

$$I(u+x, v+y) \approx I(u,v) + I_x(u,v)x + I_y(u,v)y \quad (1)$$

2. Approximation by a Taylor expansion.

$$S(x, y) \approx \sum_u \sum_v w(u, v) (I_x(u, v)x + I_y(u, v)y)^2 \quad (2)$$

3. Construction of weighted local gradients in matrix form, where I_x and I_y are partial derivatives of I in the x and y directions.

$$A = \sum_u \sum_v w(u, v) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \langle I_x^2 \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y^2 \rangle \end{bmatrix} \quad (3)$$

4. Choosing the point with two "large" eigenvalues a and b , because a corner is characterized by a large variation of S in all directions of the vector $[x,y]$.

5. If $a=0, b=0$, then the pixel (x,y) has no features of interest.

If $a=0, b \gg 0$, the point is counted as an edge.

If $a \gg 0, b \gg 0$, a corner is found.

However, the eigenvalues computation is computationally expensive, since it requires the computation of a square root. In the commercial robotics world, Harris corners are used by state-of-the-art positioning and location algorithms, various image-processing algorithms for asset tracking, visual odometry and image stabilization.



Fig. 1. Original image and Harris corner key points.

B. SIFT (Scale Invariant Feature Transform)

The most popular method for key point extraction is SIFT. Features are invariant to image scaling, translation, rotation, partially invariant to illumination changes, and affine transformations or 3D projection. It uses Differences of Gaussians (DoG) with fitted location, scale and ratio of principal curvatures for feature detection. These features are similar to neurons located in the brain's inferior temporal cortex, which is used for object recognition in primate vision. Features are efficiently detected through a staged filtering approach that identifies stable points in scale space. Image keys are created that allow for local geometric deformations by representing blurred image gradients in multiple orientation planes and at multiple scales.

The algorithm can be described in following steps.

1. The convolution of image and Gauss filter is made with different σ values.

$$\eta(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (4)$$

where k – scale coefficient, and $*$ - convolution. The candidates for key points are formed by $D(x,y,\sigma)$ extremal points calculation.

$$D(x,y,\sigma) = (\eta(x,y,\sigma)) * I(x,y) \quad (5)$$

2. The points allocated along the edges are excluded with the help of Hesse matrix, calculated in candidate points of the previous step.

$$H = \begin{pmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{pmatrix} \quad (6)$$

Because of the fact that the main curving along the edges have larger values, than in case of normal direction and the fact that Hesse matrix eigenvalues are proportionat to the main curving of the $D(x, y, \sigma)$, we need only to compare the Hesse matrix eigenvalues.

3. For the rotational invariance the orientation histogram is calculated over the key point neighbourhood with chosen step. For every σ the algorithm finds the orientation histogram extremal values.

$$\Theta(x, y) = \arctg \frac{L(x, y+1) - L(x, y-1)}{L(x+1, y) - L(x-1, y)} \quad (7)$$

$$L(x, y) = \eta(x, y, \sigma) * I(x, y) \quad (8)$$

For invariant description of the key point the following algorithm is used.

1. Choosing the neighbourhood around the key point.
2. Calculation of the gradient value in the key point and its normalizing.

The neighbourhood describing salient feature pattern is formed with the help of the replacemtn of a gradient vector by the number of its main components. It conduces to the salient feature number reduction and the affine transformation invariance is achieved, because the first main components are located along the main axes in computed gradients space. Fig. 2 illustrates the result of SIFT key point detection.



Fig. 2. Key points detection and matching using SIFT method

The major disadvantage of SIFT is that the algorithm takes too long to run and computationally expensive. In some cases it produces too few features for tracking.

C. SURF (Speeded Up Robust Features)

Another useful method of key point extraction is SURF (Speeded Up Robust Features) [1]. The descriptor comes in two variants, depending on whether rotation invariance is desired or not. The rotation invariant descriptor first assigns an orientation to the descriptor and then defines the descriptor within an oriented square. The other version, called U-SURF, for Upright-SURF, which is not rotation invariant, simply skips

the orientation assignment phase. In this method the search of key point is made with the help of Hesse matrix.

$$H(f(x, y)) = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}, \quad (9)$$

$$\det H = \frac{\partial^2 f}{\partial x^2} \frac{\partial^2 f}{\partial y^2} - \left(\frac{\partial^2 f}{\partial x \partial y} \right)^2.$$

The Hessian is based on LoG (Laplacian of Gaussian) using the convolution of pixels with filters. This approximation of Laplacian of Gaussian is called Fast-Hessian.

The Hessian reaches an extreme in the points of light intensity gradient maximum change, that's why it detects spots, angles and edges very well. Hessian is invariant to the rotation, but not scale-invariant. For this reason SURF uses different-scale filters for Hessian finding.

The maximum light intensity change direction and the scale formed by Hesse matrix coefficient are computed for each key point. The gradient value is calculated using Haar filter (Fig. 3).



Fig. 3. Haar filters

For effective Hesse and Haar filter computation image integral approximation is used.

$$H(x, y) = \sum_{i=0, j=0}^{i \leq x, j \leq y} I(i, j) \quad (10)$$

where $I(i, j)$ — light intensity of image pixels.

After key points are found, SURF algorithm forms the descriptors. Descriptor consists of 64 or 128 numbers of for each key point. These numbers display a fluctuation of a gradient near a key point. The fact that a key point is a maximum of Hessian guarantees the existence of the regions with different gradients [1]. Fig. 4 illustrates the results of SURF key point detection.

The rotation invariance is achieved, because gradient fluctuations are calculated by the gradient direction over the neighborhood of a key point. The scale invariance is achieved by the fact that the size of the region for descriptor calculation is defined by the Hesse matrix scale. Gradient fluctuations are computed using Haar filter.

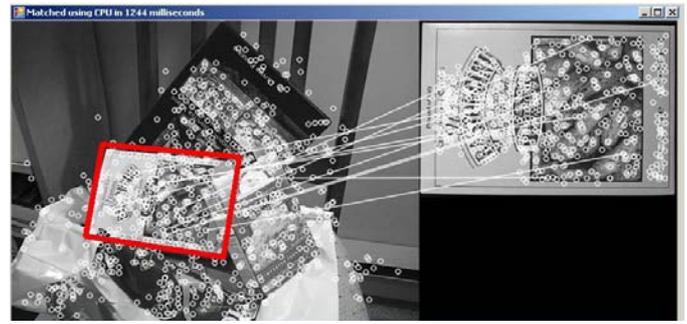


Fig. 4. SURF method key points detection.

SURF approximated, and even outperformed, previously proposed schemes with respect to repeatability, distinctiveness, and robustness. SURF also computed and compared much faster than other schemes, allowing features to be quickly extracted and compared. But for some classes of images with homogeneous texture it shows low level of key points matching precision.

III. KEY POINTS DESCRIPTORS

For detected features matching we need key points descriptors. Key point descriptor is a numerical features vector of the key points neighborhood.

$$D(x) = [f_1(w(x)) \dots f_n(w(x))] \quad (11)$$

Feature descriptors are used for making the decision of images identity. The simplest descriptor is a key point neighborhood itself.

The major property of any feature matching algorithm is distortion varieties, which an algorithm can manage with. The following distortions usually are considered:

- 1) *scale change* (digital and optical zoom, movable cameras etc.);
- 2) *image rotating* (camera rotating over the object, object rotating over the camera);
- 3) *luminance variance*.

A. Scale change invariance.

While using scale-space feature detector it can be possible to achieve scale change invariance. Before descriptor calculation normalizing is held according to feature local scale. For example, for the scale-space coefficient of 2 we need to scale the feature neighborhood with the same value of scale coefficient.

If descriptor consists of equations only with normalized differential coefficients, space scaling operation is not necessary. It is sufficient to calculate differential coefficients for the scale associated with the feature.

B. Rotating invariance.

The simplest way to achieve rotating invariance is to use descriptors formed of rotating invariant components.

The major disadvantage of such an approach lies in the fact that it is impossible to use components with rotating dependence, but the amount of rotating invariant components is restricted.

The second way to achieve the rotating invariance is previous key point neighborhood normalizing for rotate compensation. For key point neighborhood normalizing we need feature orientation estimation. There are a lot of feature local orientation estimation methods, but all of them are connected with feature neighborhood gradient direction calculation. For example, in SIFT method the rotation invariance is achieved as follows.

- 1) All gradient directions angles from 0 to 360 degrees are divided into 36 equal parts. Every part is associated with a histogram column.
- 2) For every point from the neighborhood a phase and a vector magnitude are calculated.

$$\text{grad}(x_0, \delta) = (L_{x, \text{norm}}(x_0, \delta) L_{y, \text{norm}}(x_0, \delta)) \quad (12)$$

$$\Theta = L_{y, \text{norm}}(x_0, \delta) / L_{x, \text{norm}}(x_0, \delta) \quad (13)$$

$$A = |\text{grad}(x_0, \delta)| \quad (14)$$

$$H[i_\Theta] = H[i_\Theta] + Aw \quad (15)$$

where i – index of gradient phase cell, w – weight of a point. It can be possible to use the simplest weight of 1 or use Gaussian with the center in point a .

- 3) After that for every key point neighborhood direction $\varphi = i * 10^\circ$ is chosen, where i is index of maximum from histogram elements. After orientation calculation the normalizing procedure is produced. A key point neighborhood rotates over the neighborhood center. Unfortunately, for some features orientation becomes wrong, and that descriptors cannot be used in further comparison. For every point from the neighborhood a phase and a vector magnitude are calculated.

C. Luminance invariance

For luminance invariance measurement we need the model of image luminance. Usually an affine model is used. It considers the luminance of the pixels changes according to the rule:

$$I_L = a * I(x) + b \quad (16)$$

This luminance model doesn't conform to real actuality correctly, and the luminance processes are much more complex, but it is sufficient for small local regions luminance representation.

According to affine luminance model to avoid luminance influence on pixels values in the key point neighborhood.

$$I_{\text{mean}}(w(x)) = I(w(x)) - \text{mean}(I(w(x))) \quad (17)$$

$$I_{\text{result}}(w(x)) = I_{\text{mean}}(w(x)) / \text{std}(I(w(x))) \quad (18)$$

where $\text{mean}(I(w(x)))$ and $\text{std}(I(w(x)))$ denote sample average and mean square deviation in neighborhood of w , $I_{\text{mean}}(w(x))$ – the translated neighborhood and $I_{\text{result}}(w(x))$ – the resulting neighborhood, which must be used for luminance invariance calculation.

IV. WAVELET TRANSFORMATION BASED KEY POINT DETECTION

Another way of key points extraction is using of discrete wavelet transformation. Discrete wavelet transformation produces a row of image approximations. For image processing Mall algorithm is used. The initial image is divided into two parts: high frequency part (details with sharp luminance differences) and low-frequency part (smoothed scale down copy of the original image). Two filters are applied to the image to form the result. It is an iterative process with the scaled down image copy as the input.

A. Discrete Wavelet Transformation

Wavelet transformation is rather new direction of theory and technique of signal, image and time series processing. It has been discovered at the end of the XX century and now is used in different spheres of computer science such as signal filtration, image compression, pattern recognition etc. The reason of its widely spread using is based on wavelet transformation ability of exploring inhomogeneous process structure.

Discrete wavelet transformation produces a row of image approximations. For image processing Mall algorithm is used (Fig. 5). The initial image is divided into two parts: high frequency part (details with sharp luminance differences) and low-frequency part (smoothed scale down copy of the original image). Two filters are applied to the image to form the result. It is an iterative process with the scaled down image copy as the input. [5]

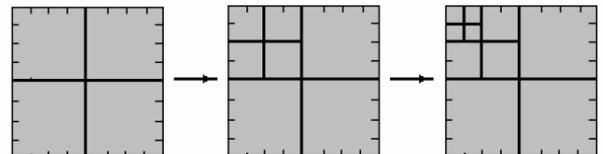


Fig. 5. Mall algorithm

B. Key Points Detection

Wavelet image transformation can be used for key points detection. The saliency of the key point is formed by the weights of wavelet coefficients. [3]

In the method proposed in this article the key point extraction algorithm calculates the weight of every image pixel using the following equation:

$$C_i(f(x,y)) = \sqrt{dh_i^2(x,y) + dv_i^2(x,y) + dd_i^2(x,y)} \quad (19)$$

where $C_i(f(x,y))$ – the weight of the point on the level i of detalization, $dh_i(x,y)$ – horizontal coefficient on the level i , $dv_i(x,y)$ – vertical coefficient on the level i , $dd_i(x,y)$ – diagonal coefficient on the level i . At the first step all weights are equal to zero. Then wavelet transformation is carried out until it reaches the level n . Each rather large wavelet coefficient denotes a region with a key point of the image. Weight is calculated using the following formula (19) then recursive branch is exercised.

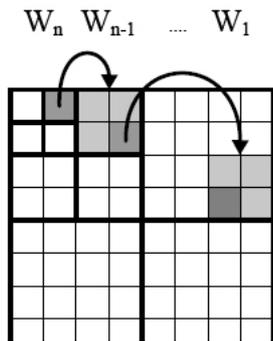


Fig. 6. Key point weight calculation.

This algorithm repeats for all decomposition levels. The final value of the weight of pixel is formed by the wavelet coefficients of previous levels. After key points sorting the point larger than the desired threshold are chosen.

In the image on Fig. 1 and Fig. 4 the key points are detected using Harris detector and wavelet based method. In case of Harris detector key points are located in the corners and have little dispersion over the image. In the case of wavelet based method the image is covered with key points proportionally.

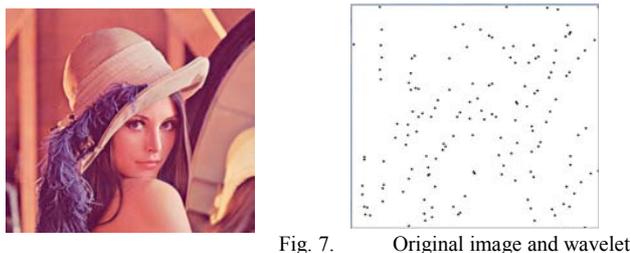


Fig. 7. Original image and wavelet based key points.

C. Key Points Descriptor

For points matching we need to create a descriptor, which can describe the point and tell the differences between them. SIFT and SURF descriptors are based on pixels luminance over the region in the point neighborhood. In this paper we offer using the wavelet coefficient, which are produced from the luminance are stable for various luminance changes.

The descriptor is formed from the pixel wavelet coefficients, received from the wavelet decomposition of key point neighborhood. Each neighbor is characterized by 4 wavelet coefficients: the base coefficient, horizontal, diagonal and vertical ones. The dimension of in the descriptor is fixed on $4*16$. The size of the neighborhood region depends on the size of image and the wavelet decomposition level. The experiments have shown that it is possible to use the depth of wavelet decomposition equal or greater than 3. For example, for the region size of 64 neighbors we need the 3rd decomposition level to form the descriptor of $4*8$ and in the case of the dimensionality of neighborhood increase we should also increase the level of transformation. Experiments have shown that such an increase takes more time for computation, but in some cases it allows to avoid matching errors.

Fig. 8 illustrates wavelet based key point extraction and matching result. The software application was implemented with the use of C# programming language in Microsoft Visual Studio 2008.

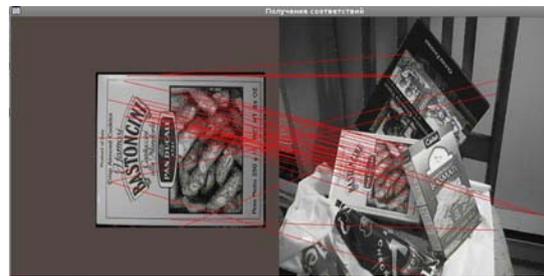


Fig. 8. Key points detection and matching using wavelet based method

D. Segmentation using wavelet key points

Wavelet based key points detection algorithm can be used for image segmentation. On the first step wavelet based key points retrieval is carried out. All key points are marked with black color, and other points of the image are marked with white color. Then connected components retrieval algorithm is applied. This algorithm considers the black color of key points as the background and the white color of ordinary points as objects on the foreground. After that key points spaceless sequence surrounded pixels joining is produced. For different segments marking the algorithm of connected components line-by-line marking is used.

Described above segmentation algorithm is computationally efficient. It can be used in systems with restricted resources. The computational efficiency is reached because of the fact that this algorithm finds only large objects on the image. Wavelet transformation explores an image on different scales and finds only the points, which have saliency on all levels. This property is owned by the points with major luminance change. In resulting image we “loose” all little components and see only the larger ones. Fig. 9 shows the segmentation results produced. The software application was implemented with the use of C# programming language in Microsoft Visual Studio 2008 and was evaluated on mobile devices with restricted resources. (The photos are made of the mobile device emulator on PC.)



Fig. 9. Segmentation result on mobile device: a –original image, b – wavelet based key points, c – segmented image.

E. Future Work

Future work will be referred to the improvement of the proposed method of wavelet based key point detection. It is necessary to increase the accuracy of key point matching and to decrease the computational complexity of descriptor finding.

For another thing we need more experiment results for detecting rotation, scale and luminance invariance of the proposed method.

REFERENCES

- [1] H. Bay, A. Ess, T. Tuytelaars, and L. Gool, "SURF: Speeded Up Robust Features," *Computer Vision and Image Understanding*, vol. 110, no. 3, pp. 346-359, 2008.
- [2] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in
- [3] *Proceedings of the 4th Alvey Vision Conference*, 1988, pp. 147-151.
- [4] E. Loupias, N. Sebe. "Wavelet-based Salient Points: Applications to Image Retrieval Using Color and Texture Features." *Lecture Notes in Computer Science*.
- [5] D. G. Lowe, "Object Recognition from Local Scale-Invariant Features," in *Seventh IEEE International Conference on Computer Vision*, vol. 2, Kerkyra, Greece, 1999, pp. 1150-1157.
- [6] E.J. Stollnitz, T. DeRose, and D. Salesin, "Wavelets for computer graphics - theory and applications", presented at *The Morgan Kaufmann series in computer graphics and geometric modeling*, 1996, pp.1-245.

Voice Control of Robots and Mobile Machinery

Ruslan Sergeevich Shokhirev

Institute of Technical Cybernetics and Informatics,
Kazan State Technical University,
Kazan, Russia
ruslan.shohirev@gmail.com

Abstract—*I develop a system of Russian voice commands recognition. Wavelet transformation is used to analyze the signal key characteristics. Kohonen neural network is used to recognize spoken sound based on these characteristics. Besides, I'll give a brief overview of the current state of the problem of speech recognition*

Keywords—*speech recognition; voice control; wavelet; neural network*

I. INTRODUCTION

One of the ways to improve the human-machine interaction is using of voice control interface. This approach allows to control activities of the technical devices in situations where the operator's hands are busy another work, as well as people with disabilities. In addition, this approach can be used to improve ease of use device.

There are many approaches to solving the problem of voice control at the present moment. There are many speech recognition systems in Russia and in the world. The main problems of modern Russian language recognition systems include the following:

- 1) Phonetics and semantics of the Russian language be formalized much worse compared with the English language.
- 2) There has been a little research and produced a few works on the subject of speech recognition in Russia since the USSR. This complicates the task of creating systems of recognition, because there is no well documented theoretical basis and description of modern approaches to solving this problem^[1].
- 3) Existing systems that recognize the Russian language are often built on the principle of client-server, which makes them dependent on availability and quality of communication from global network of Internet. In addition it often puts the user in relation to corporations that own these servers. This is not always possible from the point of view of safety.

The most popular speech recognition systems today can be called the client-server solutions from the corporations Google and Apple: Google Voice Search and Apple Siri. These systems are similar in their work and are based on distributed cloud computing made in corporate date-centers. Systems have extensive vocabularies in different languages, including Russian. The number of recognizable words by Google is hundreds of billion^[2]. The main application of these systems is

mobile devices and gadgets. Disadvantages are the dependence on the Internet and corporate data centers.

Both foreign and Russian companies are currently engaged in a number of studies related to speech recognition. However, to date there is no public system of Russian speech recognition.

II. STATEMENT OF THE PROBLEM

One of the applications of speech recognition systems is the control of mobile machines. At present, manual data input from the keyboard, and specialized controllers – joystick are widely used to interaction with mobile machinery. However, there are situations where it is impossible or inefficient to use these interfaces for control. Operator's hands may be busy doing other work. For example, voice commands can be used to control external video cameras during outdoor work on the space station, while the operator's hands are operating the manipulators. Just such systems can be used to control various household devices by people with limited physical abilities. In such systems the reliability speech recognition and independence of the system from external factors plays an important role, even at the expense of the number of recognizable words. On the other side the recognition of spontaneous speech does not required for these systems. They are used to enter predefined control commands in most cases.

Thus had the following research objectives:

- 1) The developed system must be autonomous and independent. I.e. all calculations related to the speech recognition must be made directly on the device, or on the local server.
- 2) The developed system should have a limited vocabulary of recognizable words. The system must be universal, namely: adding and removing commands must be performed as quickly as possible.

III. COMPOSITION OF SPEECH RECOGNITION SYSTEMS

A. General Scheme

In the general case speech recognition system (SRS) can be represented by scheme in figure 1^[3]. But some units may be missing or combined into one in real SRS. SRS that used to control some devices requires a limited set of commands, and we can use more simple scheme (figure 2) for our system.

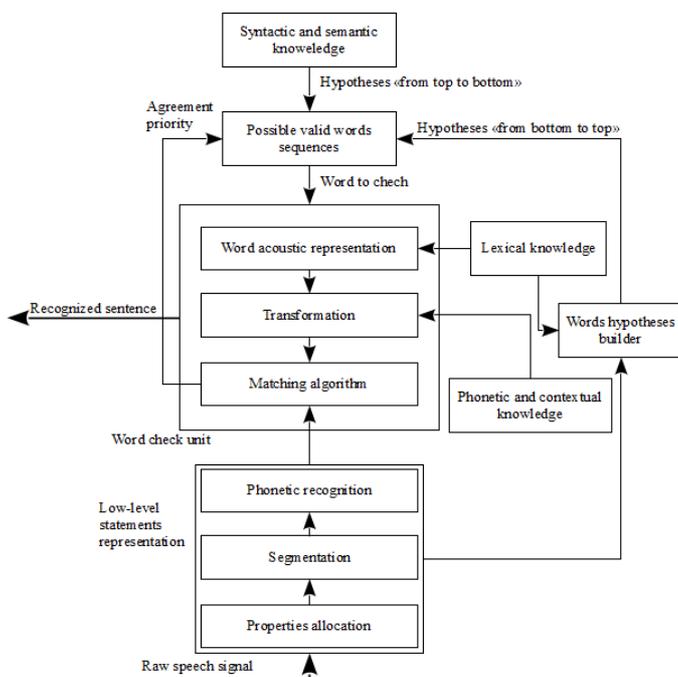


Fig. 1. SRS Common scheme

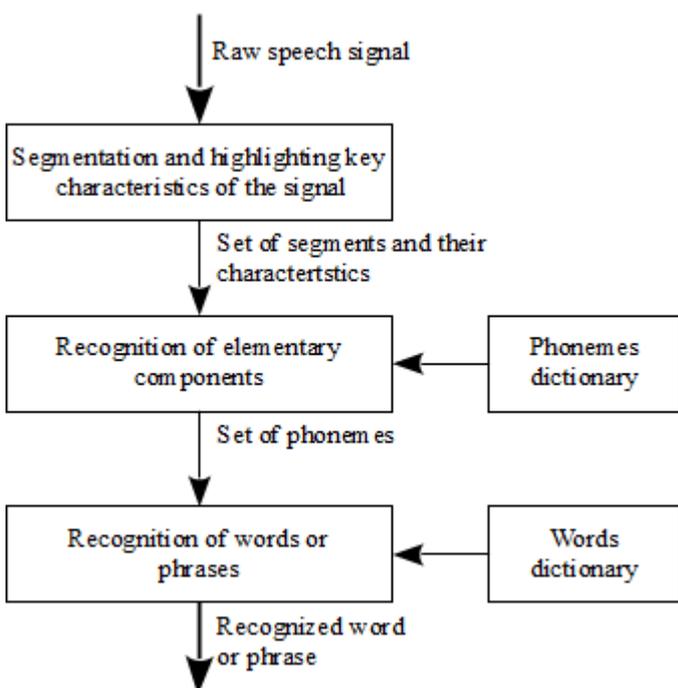


Fig. 2. Command recognition system scheme

Auxiliary algorithms for pre-filtering and system learning are also used in addition to these basic steps.

The second part is often skipped in voice commands recognition and whole words are recognized at once. Advantages of this approach is reducing the number of calculations. But retraining such system for recognition new commands will take more time than retraining system which recognizes phonemes. Because in the second case phonemes of

which consists the new command is already in the system database, and we only need to train it to identify a new order of phonemes.

B. Selection of Signal Characteristics

Frequency spectrum changing in time is the natural characteristic of speech signal. The human brain recognizes speech exactly based on its time-frequency characteristics. Correct identification of the signal characteristics is extremely necessary for successful speech recognition. There are many approaches to solve this problem:

- Fourier spectral analysis.
- Linear prediction coefficients.
- Cepstral analysis.
- Wavelet analysis.
- And other.

Wavelet is a mathematical function that analyzes different frequency components of data. Graph of the function looks like a wavy oscillations with amplitude decreases to zero away from the origin. However, this is particular definition. Generally, the signal analysis is performed in the plane of the wavelet coefficients. Wavelet-coefficients are determined by the integral signal transformation. The resulting wavelet spectrogram clearly ties spectrum of various characteristics of the signals to the time^[4]. This way they are fundamentally different from the usual Fourier spectra. This difference gives the advantage of wavelet transformation in the analysis of speech signals that non-stationary in time.

The wavelet transformation of the signal (DWT) consistently selects more and more high-frequency parts, thus breaking the signal into several levels of wavelet coefficients. The coefficients on the first levels are the lowest frequency signal. These coefficients give a good frequency resolution and low time resolution. The coefficients on the last levels of decomposition are the highest frequency of the signal. They give good time resolution and low frequency resolution.

Thus, selection of the signal characteristics using wavelet analysis is transformation of signal into wavelet-coefficients and calculation of average values of these coefficients at each level of the wavelet decomposition.

Segmentation of the signal on phonemes is performed at this stage. A phoneme is the minimal unit of the sound structure of language. DWT can solve this problem. The signal is changing on many decomposition-levels at once in transition between phonemes. Thus, the determination of the phonemes boundaries can be reduced to finding the moments of the wavelet-coefficients changing in most of the decomposition-levels^[5].

First signal is divided into overlapping regions (frames), each of which applies DWT. We can calculate energy for each frame i and decomposition-level n :

$$E_n(i) = \sum_{j=1}^{2^n-1} d_{n,j+2^{n-1}}^2 \quad (1)$$

The signal energy (1) rapidly changes from frame to frame for each level. This is due to unavoidable noise during speech signal recording. We define E'_n to smooth energy changes. For this we replace value of E_n in window of 3 – 5 frames on the maximum value of E_{max} in this window. We calculate derivative R to determine the rate of energy change. The transition between phonemes are characterized by small and rapid changes of energy level at one or more decomposition-levels. Thus, criterion of the phonemes boundary finding is fast change of the derivative at a low energy level^[6].

C. Recognition of Phonemes

The recognition result depends on the correct identification of the detected phonemes in many respects. However, the solution of this task is not trivial. Person never pronounces sounds the same. Pronunciation depends on physical health of speaker and his emotional state. Therefore it is impossible to identify phoneme simply comparing its characteristics with the characteristics of the standard phoneme. However, all versions of pronouncing the same phoneme will somehow resemble the standard pronunciation. In other words, they will be around in the signal characteristics domain. Identification of the pronounced phoneme can be reduced to solving the problem of clustering.

Clustering of phonemes in the developed system uses a network of vector quantization based on Kohonen neural network^{[7][8]}. The advantage of neural network over k-means algorithm is that it less sensitive to outliers as it uses universal approximator – neural network.

Kohonen neural networks is a class of neural networks, their main element is the Kohonen layer. Kohonen layer consists of adaptive linear combiners. Typically, the output signals of Kohonen layer are processed by the rule “winner takes all”: the largest signal is converted into one, others in zeros. Problem of vector quantization with k code vectors W_j for a given set of input vectors S is formulated as a problem of minimizing the distortion in encoding. The basic version Kohonen network uses the method of least squares and distortions D is given by:

$$D = \sum_{j=1}^k \sum_{x \in K_j} \|x - W_j\|^2$$

where K_j is consists of those points of $x \in S$, which are closer to W_j than to other W_l ($l \neq j$). In other words, K_j consists of those points $x \in S$, which are encoded code vector W_j . Set S is not known when the network not configured to the speaker. In this case online method is used to learn network. Input vectors x are processed one by one. The nearest code vector (a “winner” who “takes all”) $W_j(x)$ is sought for each of them. After that, this code vector is recalculated as follows:

$$W_{j(x)}^{new} = W_{j(x)}^{old} (1 - \theta) + x \theta$$

where $\theta \in (0, 1)$ is learning step. The rest of the code vectors do not change in this step. The online method with fading rate of learning is used to ensure stability: if T is the number of steps of training, then we put $\theta = \theta(T)$. Function of $\theta(T) > 0$ is chosen so that the $\theta(T) \rightarrow 0$

monotonically as $T \rightarrow \infty$ and the series $\sum_{T=1}^{\infty} \theta(T)$ diverges such, $\theta(T) = \theta_0 / T$.

D. Recognition of Words

After receiving the sequence of phonemes from the original signal we must map this sequence to voice command in the system database or indicate that the spoken word is not recognized. However, this problem is also a non-trivial. Differences in the pronunciation of sounds can be so significant that the same sound pronounced by a person will be identified by the system as two entirely different phonemes. Thus, only based on comparison the sequence of spoken phonemes to the standard sequence of phonemes of command, we can not say that this or that command was pronounced. One of solutions this problem is using of algorithm for finding the shortest distance between spoken word and standard system commands.

In the developed system Levenshtein distance (edit distance) is used as a measure of distance between the words. The Levenshtein distance is a string metric for measuring the difference between two sequences^[9]. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertion, deletion, substitution) required to change one word into the other. Mathematically, the Levenshtein distance between two strings a, b is given by $lev(a, |b|)$ where

$$lev(i, j) = \begin{cases} \max(i, j) & , \min(i, j) = 0 \\ \min \begin{cases} lev(i-1, j) + 1 \\ lev(i, j-1) + 1 \\ lev(i-1, j-1) + [a_i \neq b_j] \end{cases} & , \text{else} \end{cases}$$

In this case, the characters is a phoneme, the source string is pronounced sequence of phonemes, and the resulting string is a sequence of phonemes in the standard command

IV. CONCLUSION AND FUTURE WORKS

At the moment, I realize algorithms described above in Matlab environment. The most immediate problem is study of selected algorithms efficiency and taking action to improve it. Here are some possible directions for improving the system:

- Using of pre-filtering algorithms.
- Experiments on the choice of the most suitable wavelet for speech processing.
- Check the efficiency of the wavelet packet analysis instead of the usual.
- Check the efficiency of the Kohonen neural in comparison with the other clustering algorithms.
- Check of efficiency of other algorithms to determine the distance between the spoken word and standards.
- Assessing the impact of size and composition of the commands dictionary on the system performance.

Later, algorithms tested in Matlab environment will allow us to develop software system in the C++ language. After that I

will be able to make field testing of the system in controlling educational mobile robot.

REFERENCES

- [1] Nitrov M. “Распознавание русской речи: состояние и перспективы” in “Речевые технологии”, vol.1, 2008, pp. 83-87.
- [2] M. Pinola “Speech Recognition Through the Decades: How We Ended Up Siri” article on PCWorld web-site, 2011. URL: <http://www.pcworld.com>
- [3] Li U. “Методы автоматического распознавания речи”, vol.1, vol.2, Moscow, “Наука”, 1983.
- [4] Daubechies I. “Ten Lectures on Wavelets”, SIAM, 1 edition, 1992.
- [5] Ermolenko T., Shevchuk V. “Алгоритмы сегментации с применением быстрого вейвлет-преобразования” Papers accepted for publication on the website of the international conference “Диалог”, 2003. URL: <http://www.dialog-21.ru>
- [6] Vishnjakova O., Lavrov D. “Автоматическая сегментация речевого сигнала на базе дискретного вейвлет-преобразования” in “Математические структуры и моделирование” vol. 23, 2011, pp. 43-48
- [7] Tan Keng Yan, Colin “Speaker Adaptive Phoneme Recognition Using Time Delay Neural Networks” National University of Singapore, 2000
- [8] Hecht-Nielsen R., “Neurocomputing”, Reading, MA: Addison-Wesley, 1990
- [9] Levenshtein V. “Двоичные коды с исправлением выпадений, вставок и замещений символов”. Доклады Академий Наук СССР 163 (4), pp. 845–8, 1965.

Service-oriented control system for a differential wheeled robot

Alexander Mangin, Lyubov Amiraslanova, Leonid Lagunov, Yuri Okulovsky

Ural Federal University
Yekaterinburg, Lenina str. 51
Email: yuri.okulovsky@gmail.com

Abstract—Double-wheeled robot is a classical yet popular architecture for a mobile robot, and many algorithms are created to control such robots. The main goal of the paper is to decompose some of this algorithms into services in service-oriented system with an original messaging model. We describe data types that are common for robotics control, and ways to handle them in .NET Framework. We bring a list of various services' types, and each of them can be implemented in several ways and linked with other services in order to create flexible and highly adjustable control system. Service-oriented systems are scalable, can be distributed on many computers, and provides huge debugging capacities. The service-oriented representation is also very useful when teaching robotics, because each service is relatively simple, and therefore algorithms can be presented to students gradually. In this paper, we also focus on a particular services' types, which provides the correction of the robot by the feedback, describe the original algorithm to do so, and compare it with several others.

Index Terms—robotics, service-oriented approach, double-wheeled robots

INTRODUCTION

A differential wheeled robot is a mobile robot whose movement is based on two separately driven wheels, placed on either side of the robot body. Examples of this architecture are Roomba vacuum cleaner [2], Segway vehicle [6], various research and educational robots (e.g., [7]).

Differential wheeled robot is a very simple and effective architecture, both in mechanic and control means, and many various algorithms were developed to control it. In this paper we decompose some of these algorithms within the service-oriented approach. In SOA, the functionality of the program is decomposed into a bunch of services, which communicate by TCP/IP protocol, or by shared memory, or by other means. Each of the services performs a single and simple task, and provides some result in response to an input in a contract-defined format.

Service-oriented approach is widely used in robotics [12], [17]. Its main advantages are as follows. The system can be distributed among several computers, which is important, because the real robotics is very resource-expensive. The system is also decentralized, which allows it to operate even if some auxiliary parts stop working due to errors. The service-oriented approach also has a great value in education. The service-oriented decomposition allows making a step-by-step acquaintance with complex algorithms, by dividing them to small and well-understandable parts, therefore simplifying teaching

the algorithms to students. The decomposition also facilitates research and development. While running the algorithm, all the information that passes between services can be stored in logs and then viewed, which offers a great debugging feature. Also, modern development techniques, like agile development, become more applicable, because the parts that the algorithm is divided into can be distributed between developers, can evolve gradually, and can be thoroughly tested with unit and functional tests.

Overall, service-oriented approach to robot's control is one of the most popular and promising. Many control system are founded on it, and most prominent are Microsoft Robotics Developer Studio [3] and Robotic Operating System [10], [5]. In [11] we propose RoboCoP, a Robotic Cooperation Protocol, which introduces an innovative messaging model into service-oriented robotics. In RoboCoP, services have inputs and outputs, which are interconnected in a strict topology. For example, when analyzing images, a Camera services output is plugged in to a Filter services input, and the Filter in turn is connected to a Recognizer service in the same way. So the signal propagates along the control system from service to service, and is subsequently processed by them. This messaging model is used in LabView [18], DirectShow [19] and other software, but is new for robotics. For example, in MRDS, services exchange messages via a central switch, in ROS they use broadcast messaging model, etc. [11].

In [11], we implemented this new messaging model for interconnection of independent applications with an open and simple protocol. We also built a control system for a manipulator's control, and therefore assert the effectiveness of our approach. In this paper, we bring another example of decomposition into RoboCoP services, this time for the control system for a differential wheeled robot. All the services and algorithms, described in the paper, are implemented. The system was tested on the real differential-wheeled robots during the International contest on autonomous robots control "Eurobot" [1].

In section I, we describe the data types that are important for control of the differential wheeled robot. We also describe an innovative LINQ-style [15] approach to their processing. In section II, we bring the service-oriented control systems for differential wheeled robots, and in section III we explore the peculiarities of some used algorithms.

I. PRESENTING AND PROCESSING ROBOTICS DATA

A. Data types

Service-oriented control system consists of services, which transform the information from one type to another. In this section we describe the important data types in our system. The most fundamental structure is a differential wheeled movement (DWM), which is a tuple $(v_{0,l}, v_{0,r}, v_{1,l}, v_{1,r}, T)$ where $v_{0,s}$ are linear speeds of left ($s = l$) and right ($s = r$) wheels at the beginning of movement, $v_{1,s}$ are speeds at the end of movement, and T is a time the movement lasts. One DWM describes the movement with constant acceleration of wheels, which is a reasonable physical model of real engines.

Let a_s be an acceleration of the corresponding wheel, $a_s = \frac{v_{1,s} - v_{0,s}}{T}$. Let a be the linear acceleration of the robots coordinate system, $a = \frac{a_r + a_l}{2}$. Similarly, $v_s(t)$ is a speed of the corresponding wheel at the time t , so $v_s(t) = v_{0,s} + a_s t$, and $v(t)$ is the linear speed of the robot, $v(t) = \frac{v_l(t) + v_r(t)}{2}$. We can now compute a direction $\alpha(t)$ as follows

$$\alpha(t) = \frac{B_r(t) - B_l(t)}{\Delta}$$

where Δ is the distance between wheel, and $B_s(t)$ is the total path covered by s -th wheel at the time t , $B_s(t) = v_{0,s}t + \frac{a_s t^2}{2}$. The curvature $R(t)$ of the robot trajectory can be obtained as $R(t) = \frac{\Delta}{2} \frac{v_r(t) - v_l(t)}{v_r(t) + v_l(t)}$.

Let $L(t)$ be the offset of the robot at time t along the tangent of the direction at time $t = 0$, and $h(t)$ be the offset at the normal to the initial direction. It can be shown that

$$L(t) = \int_0^t v(\tau) \cos(\alpha(\tau)) d\tau$$

$$h(t) = \int_0^t v(\tau) \sin(\alpha(\tau)) d\tau$$

Depending on DWM, the robot covers trajectories of different shape.

- 1) The straight line when $v_{0,l} = v_{0,r}$ and $v_{1,l} = v_{1,r}$, and $L(t) = v(0)t + at^2/2$.
- 2) The turn at the spot when $v_{0,l} = -v_{0,r}$ and $v_{1,l} = -v_{1,r}$. In this case, $L(t) = h(t) = 0$, and $\alpha(t)$ gives the direction of the robot at the time t .
- 3) The circle arc when $v_{0,l}/v_{0,r} = v_{1,l}/v_{1,r}$, so $R(t) = R$ is constant, and $L(t) = R \cos \alpha(t)$ and $h(t) = R \sin \alpha(t)$.
- 4) The spiral arc when $a_l = a_r$ and $v_l(0) \neq v_r(0)$. Let $q(t) = (v_r(t) - v_l(t))/\Delta$, and in this case

$$L(t) = \frac{1}{q} \left[v(0) \sin tq + at \sin tq + \frac{a}{q} (1 - \cos tq) \right]$$

$$h(t) = \frac{1}{q} \left[v(0) (1 - \cos tq) - at \cos tq - \frac{a}{q} \sin tq \right]$$

- 5) The clothoid segment otherwise.

Clothoid is the most general case, and one need Fresnel integrals $S(t) = \int_0^t \sin \pi \tau^2 / 2 d\tau$ and $C(t) = \int_0^t \cos \pi \tau^2 / 2 d\tau$ to

compute the robots location. Let us consider some intermediate values:

$$X = \frac{a_r + a_l}{a_r - a_l}$$

$$v_c(t) = \frac{v_r(t) + v_l(t)}{2} + X \frac{v_r(t) - v_l(t)}{2}$$

$$\Delta_c = \Delta X / 2$$

$$U(t) = \frac{v_r(t) - v_l(t)}{2\Delta |a_r - a_l|}$$

$$\delta = \text{sign}(a_r - a_l)$$

With this definitions, we compute $L(t)$ and $h(t)$ as follows

$$L(t) = \Delta_c \sin \alpha(t) + V_c(C(t) \cos U(t) + S(t) \sin U(t))$$

$$h(t) = \Delta_c (\cos \alpha(t) - 1) + \delta V_c(S(t) \cos U(t) + C(t) \sin U(t))$$

To specify the shape the robot should cover, we use data structures, called geometries. Currently, there are three geometries available: the straight line, which is parametrized by its length; the turn on the spot, which depends on the desired angle; and the circle arc. We can set the arc by its radius and the total rotation angle, or by the radius and the total distance that should be covered. Spirals and clothoids are not implemented, because they are hard to define, and generally, by our opinion, the benefits of their use are greatly overwhelmed by the complexity of their handling.

Another important kind of data is sensors measurements. Encoders are devices that count the rotations of wheels and store the measurements in EncodersData structure. Encoders are considered the most fundamental sensors, and many algorithms use them. Accelerometers and gyroscopes are sometimes used to collect data about acceleration and direction of the robot. This data is used in the processed format of NavigatorData, which consists of the robots basis in 2-dimensional space and of the time when measurements were taken. Aside from using in control algorithms, the series of NavigatorData is also used for drawing charts.

When measurements are considered, it is convenient to introduce spans between them. NavigatorDataSpan contains a time interval between two NavigatorData, and displacement of the last basis relatively to the first one. Encoders data span contains a time interval and differences of distances performed by the left and the right wheels.

B. Conversions

Let us examine some possible conversions between the described data types. This conversions are widely used in control algorithms, to answer the questions like “where a robot, driven by a given DWM, is situated”.

DWM can be converted to NavigatorDataSpan. Backwards conversion is impossible because some displacements, e.g. a shift to the right side, cannot be achieved by the differential wheeled robot at all. DWM and EncodersDataSpan are mutually convertible.

DWM and geometries are conventionally convertible. DWM can be converted to a geometry, but since spirals and clothoids

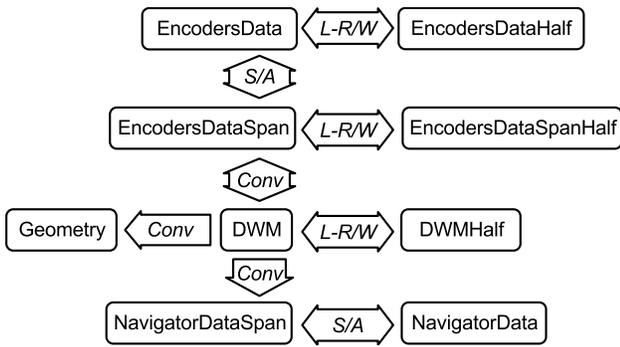


Fig. 1. A map of conversions between robotics data types

are not implemented, this conversion currently works only for a limited set of DWM. Many DWMs corresponds to one geometry, and for movement planning, we should consider velocity and acceleration limitations, the finishing speed of the robot at the previous path, and so on. Therefore, turning geometry into DWM can be done in several ways, and we should use a proper service to achieve it. However, converting geometry to some DWM is useful, and therefore we introduced a normalized DWM for lines, circles and turns. A normalized DWM is a DWM without acceleration, and has $\max(v_{0,l}, v_{0,r}, v_{1,l}, v_{1,r}) = 1$. This conversion is used only for further conversion to NavigatorDataSpan and NavigatorData, in order to draw charts for geometries.

Two measurements NavigatorData and EncodersData can be converted into a span between them of the corresponding NavigatorDataSpan and EncodersDataSpan types. A measurement and a corresponding span can be converted into the final measurement. We call this types of conversion spreading and accumulating, and they can be applied to arbitrary measurements.

Finally, we define symmetric data, i.e. data that can be naturally divided into “left” and “right” part. For example, DWM and EncodersData are symmetric. DWM can be subdivided into DWMHalf that describes the command for one wheel, and EncodersData can be subdivided into EncodersDataHalf that describes the wheels state.

The map of these methods is shown in the Figure 1, where L-R/W arrows depicts transformations of symmetric data, S/A corresponds to spreading and accumulating, and Conv denotes conversions. In Figure 1 we may see, that any data type can be turned into NavigatorData, and therefore drawn at the chart.

C. Conversions of series

Control algorithms usually deal with the series of robotics data, and so we developed means to handle such series. .NET Framework provides an incredibly powerful and convenient tool for series processing, named language-integrated queries, LINQ. An example of LINQ is shown in the Listing 1. The code in the example processes IntArray, a collection of integers. The collection is filtered with the lambda `number=>number>=10`, which maps an integer

into boolean value. With this lambda, all the integers less than 10 will be thrown off. Then the resulting collection is sorted, and converted into a collection of strings. Finally, the collection of strings is aggregated with the lambda `(stacker, str)=>stacker+", "+str`, i.e. strings are subsequently accumulated in a stacker through commas. LINQ changes the very view on how the collections should be processed, and increases enormously the codes readability and reliability.

Listing 1 The LINQ code for processing collections

```
IntArray
    .Where( number => number>=10 )
    .OrderBy( number => number )
    .Select( number=>number.ToString() )
    .Accumulate(
        (stacker, str)=>stacker+", "+str);
```

We have developed the following LINQ extensions for handling robotics data:

- 1) Conversion extensions. For example, `encSpans.ToDWM().ToNavigatorDataSpans()` gets a sequence of displacements that corresponds to initial encoders data.
- 2) Spreading and accumulating. For example, `navData.Spread().Accumulate(newBasis)` shifts the navData from initial basis to the new one.
- 3) SymmetricData handling. For example, `encData.Lefts()` and `dwms.Lefts()` give the states and commands correspondingly for the left wheel.

Our extensions are compatible with the original LINQ, for example,

```
encData.Lefts()
    .Select
        (spanHalf => spanHalf.Distance)
    .Sum()
```

gives the total distance, covered by the left wheel.

The tricky moment in our LINQ implementation is to support adequate type-inference. On other words, how the extension method `Spread` knows if it should return the sequence of EncodersDataSpan or NavigatorsDataSpan, depending on its arguments, EncodersData or NavigatorData? To do that, we developed the generic interfaces, presented in the Listing 2. The type TSpan in the method `Spread` is determined from the argument, and because NavigatorData implements interface `ISpannableDeviceData<NavigatorData, NavigatorDataSpan>`, TSpan is assign to NavigatorDataSpan. This implementation is extendable, because the method `Spread` will appear for any type that supports `ISpannableDeviceData`.

Listing 2 The hierarchy of interfaces for the correct type-inference

```

public interface
    ISpannableDeviceData<TData, TSpan>
    { ... }

public interface
    IDeviceDataSpan<TData, TSpan>
    { ... }

public class NavigatorData
    : ISpannableDeviceData
    <NavigatorData, NavigatorDataSpan>
    { ... }

public class NavigatorDataSpan
    : IDeviceDataSpan
    <NavigatorData, NavigatorDataSpan>
    { ... }

public class Helpers {
public static IEnumerable<TSpan>
    Spread<TData, TSpan>
    (this IEnumerable
    <ISpannableDeviceData
    <TData, TSpan>> data)
    { ... }
}

```

II. SERVICE-ORIENTED DECOMPOSITION OF CONTROL ALGORITHMS

In this section we offer the decomposition of a control system for a differential wheeled robot into services, which process data, described in the previous section. The whole control process is represented as a sequence of small and simple algorithms, each representing a single responsibility: creating a path for robot to go, processing data from sensors, etc. Such sequences can be best described in schematic way, as in the Figure 2. We should stress, that such decomposition is in fact half of the work while creating a service-oriented control system, because it is not easy to invent the design that looks naturally, is easy to expand and allows implementation of different kinds of control algorithms.

Here boxes are services that run simultaneously and processes data, which is depicted as arrows. The work of the control system starts, when it accepts a task from the user. The task can be expressed as a WayTask type, which is a collection of points that are to be visited by the robot. WayTask can be processed by the Pathfinder service to the collection of geometries that represents the desired path. The simplest strategy is to go from one point to another and then to turn the robot on the spot in the direction of the next point. More sophisticated versions were also developed [14].

In fact, more than one control system is depicted in the

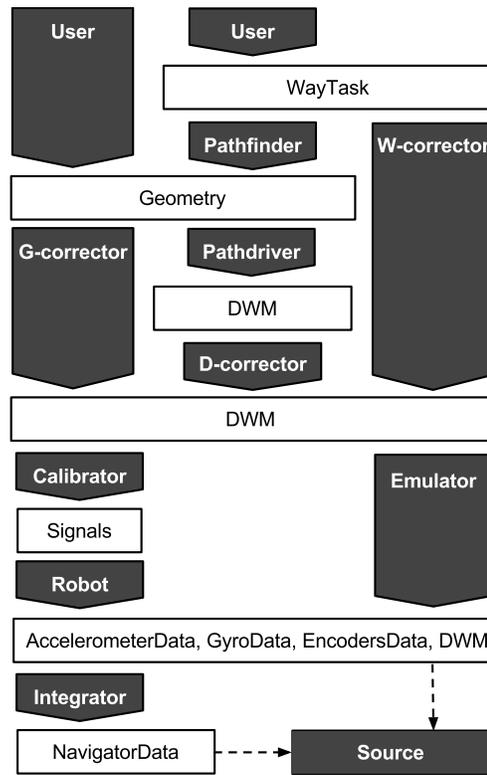


Fig. 2. Service-oriented decomposition of the control system for double wheeled robot

Figure 2, because each type of the data can be processed in many ways. Instead of going to the Pathfinder, WayTask can be used for the direct control of the robot, which is performed by the W-Corrector service. Note the important difference between W-Corrector and Pathfinder. Pathfinder completes its job at once, and we call such services “functional services”, because they are a program model of a single function. Unlike Pathfinder, when W-Corrector receives the task, it starts a continuous process of the robots control. Corrector performs an iteration for each 50-200 milliseconds, depending on its settings. At each iteration, it collects the sensors data from the Source service, which plays the role of a buffer for the measurements; then it determines the best command at the current time, and sends it further. The algorithm of W-Corrector is described in the section III.

The collection of Geometry can also be processed by its own G-Corrector. The idea of G-Corrector is that in order to follow the line, turn and circle geometries, the robot completes some given distance by one of its wheel, and travel along with the constant rate of its left and right speeds (1 for line, -1 for turn, another constant for a circle). The peculiarities of G-Correction is also described in the section III.

The collection of geometry may instead go to the Pathdriver, which converts geometries into DWM commands. Again, various strategies are possible: to stop completely after each geometry, which is very accurate but time-consuming, or to proceed to the next geometry maintaining non-zero speed. The

Pathdriver *is not* the corrector: it completes the job at once, taking a collection of geometry and producing a collection of DWM, therefore being a functional service. This collection is later processed by the D-Corrector, also described in the section III.

When the DWM is produced by some of the Correctors, it may be sent to the robot. In this case, the speeds should be converted into discrete signals, representing the duty cycles in the pulse-width modulation. This is done by the Calibrator service, which provides the correspondence between speeds and signals. The correspondence is established in the calibration process, when signals are sent to the robot, and then the rotation speed of engines is measured by encoders. Complex Calibrators may correct that correspondence while the robot is operating.

The program model of the real robot is a Robot service, which accepts discrete signals and produce the measurements of gyroscopes, accelerometers and encoders. The Robot *is not* the functional service: it does not return the measurements in response to the command. Instead, it produces measurements constantly and asynchronously. Robot service communicates with a controller board, Open Robotics [4] in our case. This board contains ATmega128 controller, slots for servos, I2C and analogous sensors, can be connected to PC via USB-UART adapter or Bluetooth adapter, and can be augmented with the amplifier board for ommutator motors. We have developed our own firmware for this board, which accepts commands in DWM format, manages servos and continuously examines sensors, collects the data and sends it to the computer in text format. It improves the built-in firmware, because sensors are monitored constantly, while built-in firmware requires sending a command to get sensors data. Other versions of Robot service can be developed for other boards and firmwares.

Instead of going to Robot, the initial DWM may be passed to Emulator, which is used for debugging. Emulator is a software that applies DWM commands to the robots current locations, computes due values of accelerometers, gyroscopes, and other sensors, adds noise into control action and feedback.

Measurements, generated by the Robot or the Emulator services, are used to get more programmer-friendly information about robots location, i.e. NavigatorData, by simple integration or Kalmans filter [8]. Alternatively, NavigatorData can be obtained directly from Emulator, but, of course, not from Robot. All measurements are stored in buffers of Source service, and when corrector starts the iteration, it reads the buffers.

III. THE CORRECTION ALGORITHMS

A. D-Correction

D-correction is the most trivial correction algorithm, and is a variation of the PID-controller [16]. At each iteration i , the due state of robot is d_i vector, which is a pair of distances, covered by the left and the right wheel. Due vectors are computed by D-corrector from the input, which is a collection of DWM, and therefore can be converted into a collection of

EncoderDataSpan. Real state of the robot can be obtained from encoders, and, assuming the work of the encoders is synchronized with iterations of D-Corrector, a serie r_i of real states on each iteration can be achieved. PID controller computes the next control action as a weighted sum of three terms. Let $e_i = r_i - d_i$, i.e. the error at the iteration i , and proportional term at the iteration k is $T_P = e_k$. Integration term T_I is defined as $\sum_{i=0}^k e_k \Delta T$, where ΔT is the time between correction iterations, and derivative term $T_D = \frac{e_k - e_{k-1}}{\Delta T}$. The resulting value $c_k = d_k + g_P T_P + g_I T_I + g_D T_D$, where g_P, g_I and g_D are the weights of corresponding terms. So $c = (c_l, c_r)$ is the state of the robot that should be achieved by the next iteration, and consists of the desired distances for both wheels. D-Corrector should now construct DWM. Let $v_{i,l}$ and $v_{i,r}$ be the end velocities of DWM, assigned at $(i - 1)$ -th iteration, $v_{0,l} = v_{0,r} = 0$. Therefore, at i -th iteration D-corrector should construct DWM with starting speed $v_{i-1,r}$ and $v_{i-1,l}$ such that this DWM covers distances c_l and c_r within the time ΔT , and therefore $v_{i,s} = \frac{2c_s}{\Delta T} + v_{i-1,s}$ for $s \in \{l, r\}$.

B. G-Correction

In D-correction, DWMs are used as a source of control actions. In G-correction, the geometries are. Suppose we need to travel along a circle, or line, or turn on the spot. The only thing needed to be done is to cover some distance L by one of the wheels, e.g. the left one, while keeping a constant rate k between the speeds of the left and right wheels. For the line $k = 1$, for the turn on the spot $k = -1$, for an arc of a circle k depends on circles radius. We have implemented G1-Correction, using encoders to get the current speed values, and PID-controller to maintain the proper value of k .

C. W-Correction

Way task is a set of points with coordinates (x_i, y_i) for $i = 1, \dots, n$. For each i we construct a vector field F_i , which indicates the proper vector of robots speed in point (x, y) while it heads toward (x_i, y_i) . Let $F_i(x, y) = (w_{i,x}(x, y), w_{i,y}(x, y))$, and

$$w_{i,x}(x, y) = k(x, y)(x - x_i)$$

$$w_{i,y}(x, y) = k(x, y)(y - y_i)$$

and $k(x, y) > 0$ is a normalizing coefficient such that

$$\begin{aligned} & \|(w_{i,x}(x, y), w_{i,y}(x, y))\| = \\ & = \min \left\{ \frac{v_{max}}{\sqrt{2} \|(x - x_i, y - y_i)\| a_{max}}, \frac{v_{max}}{\sqrt{2} \|(x - x_{i-1}, y - y_{i-1})\| a_{max}}, \right\} \end{aligned}$$

where v_{max} and a_{max} are maximum allowed speed and acceleration of the robot. This definition of F assures that if the robot is heading to the point (x_i, y_i) , and is found in the point (x, y) , it should direct to (x_i, y_i) with allowed speed, and also should be able to stop with the allowed acceleration.

When W-Corrector drives the robot to $i - th$ point and constructs the next DWM, it uses the sensors measurement to determine the current state: location (x, y) , direction ϕ ,

speeds of the left and the right wheels v_l and v_r , linear speed $v = \frac{v_l + v_r}{2}$ and torsion $q = \frac{v_l}{v_r}$. If (x, y) is close enough to (x_i, y_i) , W-Corrector increments i . Then it calculates the due speed vector $(w_x, w_y) = F_i(x, y)$, its module w and direction ψ . The W-Corrector asserts new linear speed to v , which equals to vc_v if $v < w$, and to v/c_v otherwise. Similarly, new torsion q is increased by c_q if direction ψ is on the left side from ϕ , and decreased otherwise. Finally, using v and q , speeds v_l and v_r are obtained from it, and new DWM is constructed.

D. Comparison of correction algorithms

We have implemented aforementioned algorithms and tested them to choose the best one for Eurobot competitions [1]. The preliminary results of comparison are as follows.

- D-correction is a very accurate algorithm, but is hardly compatible with electronics we possess. The problem is that in the end of movement, the small but frequent oscillations occurs, driving wheels backwards and forwards to achieve the requested position. Such oscillations puts out of action the motors amplifiers.
- W-correction is a great way to understand the control of the robot, and to visualize the correction algorithms. Still, further researches are needed to ascertain its effectiveness and obtain optimal values for its coefficients.
- G-correction is currently our best solution to correction.

IV. CONCLUSION AND FUTURE WORKS

In this paper, we presented a decomposition of a control system for a double wheeled robot into a bunch of services. We have developed the architecture of services, as well as the services themselves, and were able to use this system for control of an autonomous double wheeled robot in Eurobot 2013 competitions.

The primary direction of our future works is to introduce more correction algorithms. For example, we are developing the G-correction algorithm, which uses gyroscope data, and the services for elimination of gyroscope noise. Also, we are developing more sophisticated services for conversion of accelerometers and gyroscope measurements into NavigatorData for using it in D-Correction algorithms. Also, we plan the further decomposition of D-correction into fields generator and fields driver, and work on different sources of vector fields, such as geometries.

Other planned works in the area of double wheeled robots control includes the following topics.

- Shifting the current SOA Framework, RoboCoP, to a better solution, based on Redis [9] common memory service.
- Integrating emulator, described in [13], into the system for better visual feedback about robots location, and for getting emulated images from camera.
- Publishing the solution and some of the developed algorithms for an open access of community.
- Thorough statistical comparison for correction algorithms.

V. ACKNOWLEDGMENTS

We thank Pavel Egorov for valuable commentaries and suggestions, which help us design W-correction algorithm.

The work is supported by RFFI grant 12-01-31168, "Intelligent algorithms for planning and correction of robot's movements".

REFERENCES

- [1] Eurobot competitions. <http://eurobot.org>.
- [2] The irobot company. irobot roomba. <http://www.irobot.ru/aboutrobots.aspx>.
- [3] Microsoft robotics developer studio. <http://msdn.microsoft.com/en-us/robotics/default.aspx>.
- [4] Open robotics. <http://roboforum.ru/wiki/OpenRobotics>.
- [5] Robotics operating system. <http://www.ros.org>.
- [6] Thesegeyway company. segway. <http://www.segway.com/>.
- [7] The willow garage company. turtlebot. <http://turtlebot.com/>.
- [8] A. V. Balakrishnan. *Kalman filtering theory*. Optimization Software, Inc., Publications Division, 1984.
- [9] J. L. Carlson. *Redis in Action*. Manning, 2012.
- [10] J. L. Foote, E. Berger, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. <http://www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf>, 2009.
- [11] D. O. Kononchuk, V. I. Kandoba, S. A. Zhigalov, P. Y. Abduramanov, and Y. S. Okulovsky. Robocop: a protocol for service-oriented robot control system. In *Proceedings of international conference on Research and Education in Robotics - Eurobot 2011*. Springer, 2011.
- [12] J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22:132, 2007.
- [13] M. Kropotov, A. Ryabykh, and Y. Okulovsky. Eurosims - the robotics emulator (russian). In *Proceedings of the International (43-th Russian) Conference "The contemporary problems of mathematics"*, 2012.
- [14] A. Mangin and Y. Okulovsky. The implementation of the control system for double wheeled robot (russian). In *Proceedings of the International (44-th Russian) Conference "The contemporary problems of mathematics"*, 2013.
- [15] F. Marguerie, S. Eichert, and J. Wooley. *LINQ in Action*. Manning, 2008.
- [16] R. C. Panda, editor. *Introduction to PID Controllers - Theory, Tuning and Application to Frontier Areas*. InTech, 2012.
- [17] M. Sombly. Software platforms for service robotics. <http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Updated-review-of-robotics-software-platforms/>, 2008.
- [18] J. Travis. *LabVIEW for Everyone*. Prentice Hall, 2001.
- [19] P. Turcan and M. Wasson. *Fundamentals of Audio and Video Programming for Games (Pro-Developer)*. Microsoft Press, 2004.

Schedulling the delivery of orders by a freight train

A.Lazarev

V.A.Trapeznikov Institute
of Control Sciences
of Russian Academy of Sciences,
M.V.Lomonosov Moscow State University,
National Research University
Higher School of Economics
Email: jobmath@mail.ru

E.Musatova

V.A.Trapeznikov Institute
of Control Sciences
of Russian Academy of Sciences,
Email: nekolyap@mail.ru

N.Khusnullin

V.A.Trapeznikov Institute
of Control Sciences
of Russian Academy of Sciences,
Email: nhusnullin@gmail.com

Abstract—In this research it was considered the particular case of a railway problem, specifically, the construction of orders delivery schedule for one locomotive plying among three railway stations. In this paper it was suggested a polynomial algorithm and were shown the results of a computing experiment.

I. INTRODUCTION

Nowadays, problems of the rail planning are attracting attention of specialists due to the fact that they are challenging, tough, nontrivial and, what is more important, are of practical significance.

In this research we consider the problem of making up a freight train and the routes on the railway. It is necessary from the set of orders available at the stations to determine time-scheduling and destination routing by railways in order to minimize the total completion time.

In this paper it was studied the particular case of the problem, specifically, the construction of orders delivery schedules among 3 railway stations by one locomotive (Fig. 1). Application of dynamic programming is very effective for the solution of this problem. In this paper it was suggested a polynomial algorithm and shown the results of the computing experiment.

II. PROBLEM STATEMENT

At each station there is a set of orders available for delivery. Each order is characterized by a release date and a destination station. If the order consists of a few cars $k > 1$ then for each car there will be created a separate order.

Let us introduce the following notations:

- q – the maximal number of the cars (wagons);
- O – set of all orders;
- n – total number of orders;
- n_{ij} – set of orders available for delivery between stations i and j ;
- J_{ijk} – k -th order for delivery from station i to destination station j ;
- r_{ijk} – release time of the orders;
- p_{ij} – travelling time.

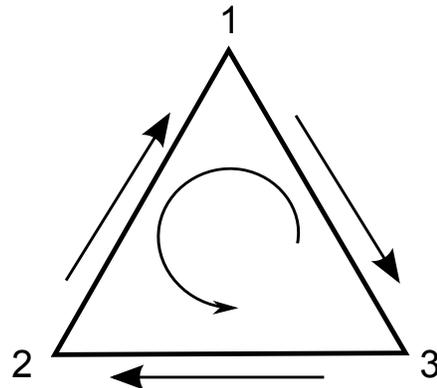


Fig. 1. The railway station location

To simplify the description of our algorithm we will assume that $p_{ij} = p \forall i \neq j$.

The objective function which tries to minimize total completion time is the following:

$$\min F = \sum_{J_{ijk} \in O} C_{ijk}, \quad (1)$$

where C_{ijk} is the completion time to destination station. Also, this function describes the average time of order delivery so it can be rewritten in the form:

$$F = \sum_{J_{ijk} \in O} \frac{C_{ijk} - r_{ijk}}{n}.$$

This problem is the generalization of the two stations problem for which polynomial algorithms are known.

It is not difficult to notice that the locomotive can have the following strategies of its route management.

1. Moving. If the locomotive stays at any station, moving is possible in one from two directions with maximum of orders available but not more than q .

2. Waiting. This point is possible if the total number of orders available for delivery is less than q (cars capacity of

a train). And this mode is impossible if the flow of the new orders is not expected.

3. Idle. This mode is necessary if the number of orders is not available for delivery. Obviously, the "idle" is impossible to use twice in succession. Also, the locomotive can idle only after departure to the station or at the starting time.

It is easy to show that using of these strategies does not cut optimal schedule that minimize the objective function. Therefore, let us assume that the locomotive movement satisfies these rules.

Definition 1. Let us suppose, that the locomotive is in the state $S(s, t, k_{12}, k_{23}, k_{31}, k_{13}, k_{32}, k_{21})$ if at the time moment $t \in T$, it is at the station s and by the current time moment has been delivered k_{12} orders from the first to the second station, k_{23} orders from the second to the third station and etc.

Let the objective function value of the state $S(s, t, k_{12}, k_{23}, k_{31}, k_{13}, k_{32}, k_{21})$ be denoted by $C(s, t, k_{12}, k_{23}, k_{31}, k_{13}, k_{32}, k_{21})$

The transition from one state to another occurs according to the strategies mentioned above. In this case, if the locomotive can move from the state S^1 to S^2 directly, then the objective value of the state can be calculated with the help of the following formula:

$$C^2 = C^1 + (t' + p) * k,$$

where t' is the time moment from the state S^1 and k is the number of the orders delivery when transforming into the new state.

The objective function value does not change if the locomotive moves to another station in idle or waiting mode.

In the case of different travelling times p_{12}, p_{23}, p_{13} the set of possible moments of the locomotive departure equals to

$$T = \{t = r_{ijk} + m_1 p_{12} + m_2 p_{23} + m_3 p_{13}\} \cup \{t = m_1 p_{12} + m_2 p_{23} + m_3 p_{13}\},$$

where $i, j \in \{1, 2, 3\}$, $k \in \{1, \dots, n_{ij}\}$, $m_1 + m_2 + m_3 \in \{0, \dots, 2n - 1\}$. It means that the power of the set T is $O(n^5)$.

III. CONCEPTS OF THE ALGORITHM

The main idea of the algorithm is the following: first of all, the graph of states in ascending order of t is built. The states are generated by the strategies mentioned above. From the same two states in the tree remains the only one that has a lower value of objective function. The solution to the problem is to reach the state which has the lowest value of the objective function from the set of the states completed:

$$\min_{s,t} C(s, t, n_{12}, n_{23}, n_{31}, n_{13}, n_{32}, n_{21}).$$

Complexity of the algorithm is estimated by the total number of states in the graph. Since the number of time moments t from the set T is $O(n^2)$, the total number of states can be estimated as $O(n^2 \prod_{i \neq j} (n_{ij} + 1))$ or $O(m^8)$, where $m = \max_{ij} n_{ij}$.

One of the key moments of our approach is the merge of the same nodes. The states are considered equal if at the time moment t both locomotives are on the same station and the numbers of the orders delivered to each station are also equal. Obviously, from two states in the tree remains the only one that has a lower value of the objective function. If the state was added to the tree before, the algorithm will replace it, otherwise we choose just added one.

This situation is represented in the Fig. 2. As you can see the value of the state $S(1,7,2,0,0,0,0,2)$ equals to 22, if its parents were enclosed in the quadrilateral and equals to 24, if its parents were enclosed in the pentagon. This condition can be an important factor in choosing between them (parent' branch). Thereby, on each step it is necessary a full tree survey.

In the simplest implementation of the algorithm the solution tree can be stored in the memory. But this approach it not optimal. For minimization of the memory used and increasing the performance this work suggests the other tree representations in the memory and also creates a garbage collector. In the RAM are stored only the states which belong to $[t_i - p; t_k]$, where t_i is the current time moment, p is the traveling time and t_k is the maximum value of the time of the set T . States that do not satisfy this condition should be relocated to the hard disk. They will be needed later when it is necessary to build and show a full branch of the tree.

During the tree creation process, as well as for the branch and bound scheme, one of the important factors is a cutting off an "unpromising" branch. We obtain the upper bound \bar{C} when the first complete state (all orders were delivered) is received.

After that, the algorithm tries to check the execution of the inequation for all of the following states in order to cut off the nodes that have the worst value of the objective function:

$$C' + \sum_{J_{ijk}} [\max\{t, r_{ijk} + p\}] > \bar{C},$$

where C' is the value of the current state, t is the current time moment. The left side of the inequation is the lower bound for the current state (all unfulfilled orders delivered to the destination after they are received immediately).

In order to illustrate our approach, let us the following example and set $n=6$, $r_{1,2}=r_{2,3}=r_{3,1}=\{1,3\}$, $q=2$, $p=2$. The locomotive at the initial time $t = 0$ is at the station 1 and has the following options:

- to stay at the station $s = 1$ until the time of order receipt $t = 1$, thus go to state $S(1, 1, 0, 0, 0, 0, 0, 0)$;
- to move to the station $s = 2$ by the idle, $S(2, 2, 0, 0, 0, 0, 0, 0)$;
- to move to the station $s = 3$ by the idle, $S(3, 2, 0, 0, 0, 0, 0, 0)$.

If at the initial time $t=0$, the locomotive stays at the station $s=1$ until the time of order receipt then it is possible to deliver the first order available either to the station $s=2$ at the next time moment, $S(2, 3, 1, 0, 0, 0, 0, 0)$, or to stay at the station $s=1$ until the time of order receipt, $S(1, 3, 0, 0, 0, 0, 0, 0)$. If at the initial time $t=0$ the locomotive moves to the station

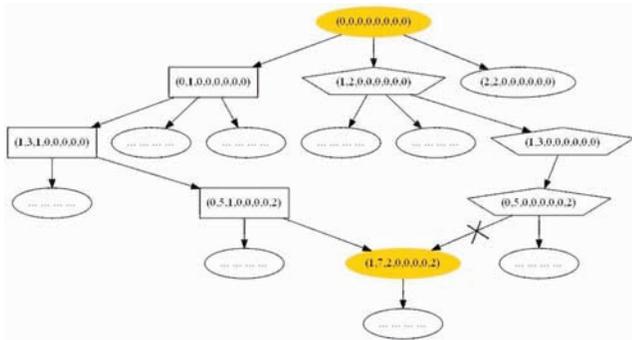


Fig. 2. The same states merging process

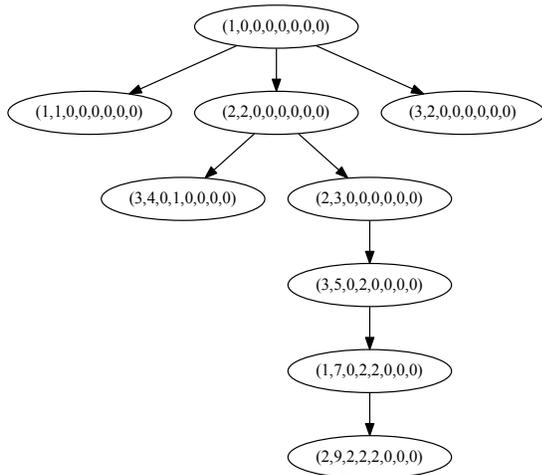


Fig. 3. The part of states graph

$s=2$ by the idle, then at the next time moment the locomotive can transport all orders available to the station $s=3$ or stay at the station $s=2$ until the time of the order receipt. In the latter case the locomotive has the only one choice: to carry all orders available at this time moment to the station $s=3$, $S(3, 5, 0, 2, 0, 0, 0)$. It should be noted that for the locomotive there are no any other options for the transition from the previous state. When the locomotive stays at the station $s=3$, he has the only one possible way: to carry all orders available to the station $s=1$, $S(1, 7, 0, 2, 2, 0, 0)$. After that the locomotive can ship remaining orders to the station $s=2$, $S(2, 9, 2, 2, 2, 0, 0)$ and in this state the locomotive delivers all orders available. The part of states graph is shown in the Fig. 3

IV. COMPUTING EXPERIMENT

Table I shows the results of a computing experiment. The first column contains input parameters – time moments, the second column contains the total number of orders, the third – the number of the nodes in the tree if the problem was solved through the blind search, the fourth – the number of theoretical nodes, in the last one - the number of the nodes which were obtained in practice. In all examples set $p = 2$, $q = 2$. Also,

TABLE I. RESULTS OF COMPUTING EXPERIMENT

input values	cars count	blind search	theoretic dynamic prgrm.	practic dynamic prgrm.
$r_{1,2} = r_{2,3} = r_{3,1} = \{1, 3\}$	6	3^{27}	648	38
$r_{1,2} = r_{2,3} = r_{3,1} = r_{1,3} = r_{3,2} = r_{2,1} = \{1, 3\}$	12	3^{51}	753	387
$r_{1,2} = r_{2,3} = r_{3,1} = r_{1,3} = r_{3,2} = r_{2,1} = \{1, 3, 5\}$	18	3^{77}	166 212	2 260
$r_{1,2} = r_{2,3} = r_{3,1} = r_{1,3} = r_{3,2} = r_{2,1} = \{1, 3, 5, 7\}$	240	3^{725}	1 154 289 852	1 268 585

from this table it may be seen that the practical complexity is much lower than it is theoretical estimation.

V. CONCLUSION

In this research it was analysed the problem of making up a freight train and its routes on the railway. Also, it was proposed a polynomial algorithm for the construction of orders delivery schedules for one locomotive plying among 3 railway stations. As an example, were represented the steps of making up a freight train and destination routing in order to minimize the total completion time. Also, there were shown the results of the computing experiments, the upper bound of the complexity and the total number of nodes while solving the problem by different approaches. The complexity of this algorithm is $O(n^8)$ operations.

Future research

- Creation of a fast and accurate technique to determine a lower bound for cutting off an unpromising branch;
- Consideration of more complex arrangement of the stations in the limits of which a locomotive will have an opportunity to deliver orders;
- Investigation of the case when orders are delivered by means of several locomotives;
- Improvement of the algorithm performance and decreasing the RAM usage;
- Parallelizing the algorithm.

REFERENCES

[Lazarev et al. "Theory of Scheduling. The tasks of railway planning"(2012)] Lazarev A.A., Musatova E.G., Gafarov E.R., Kvaratskheliya A.G. Theory of Scheduling. The tasks of railway planning. – M.: ICS RAS, 2012. – p.92

[Lazarev et al. "Theory of Scheduling. The tasks of transport systems management"(2012)] Lazarev A.A., Musatova E.G., Gafarov E.R., Kvaratskheliya A.G. Theory of Scheduling. The tasks of transport systems management. – M.: Physics Department of M.V.Lomonosov Moscow State University, 2012. – p.160

[Caprara(2011)] A. Caprara, L. Galli, P. Toth. Solution of the Train Platforming Problem Transportation Science. 2011. - 45 (2), P. 246-257.

[Zwaneveld(2001)] Zwaneveld P. J., Kroon L. G., van Hoesel S.P.M. Routing trains through a railway station based on a node packing model. European Journal of Operational Research. 2001. - No. 128 P.14-33.

- [Lazarev et al. "The integral formulation the tasks of making up a trains and their movement schedules"(2012)]
Lazarev A.A. Musatova E.G. The integral formulation the tasks of making up a trains and their movement schedules. The managing a large systems. The issue 38. M.: ICS RSA, 2012. – p.161-169.
- [Liu(2012)] Liu S.-Q., Kozan E. Scheduling trains as a blocking parallel-machine job shop scheduling problem. Computers and Operations Research. - 36(10) P. 2840-2852.
- [Baptiste "Batching identical jobs"(2000)] Baptiste Ph. Batching identical jobs. Math. Meth. Oper. Res. 2000. - No. 52 P.355-367.
- [Hagai Ilani et al. "A General Two-directional Two-campus Transport Problem"(2012)]
Hagai Ilani, Elad Shufan, Tal Grinshpoun. A General Two-directional Two-campus Transport Problem. Proceedings of the 25th European Conference on Operational Research, Vilnius, 8-11 July, 2012. - P.200.

Optimization of electronics component placement design on PCB using genetic algorithm

Zinnatova L.I.

Kazan State Technical University named after A.N.Tupolev
KSTU named after A.N.Tupolev
Kazan, Russia
Lileka-leka@yandex.ru

Suzdalcev I.V.

Kazan State Technical University named after A.N.Tupolev
KSTU named after A.N.Tupolev
Kazan, Russia
iliasuzd@mail.ru

Abstract—This article presents modified genetic algorithm (GA) of the placement of electronic components (EC) on a printed circuit board (PCB) considering criterias of thermal conditions and minimum weighted sum.

Keywords— *electronics component; printed circuit board; genetic algorithm; guillotine cutting; electronic device.*

I. INTRODUCTION

The trend of most electronic companies is towards designing more functionality but smaller packages electronic system. Therefore, solving these problems requires using a large number of criteria and constraints with high dimension initial data.

The objective is to reduce the time, increase quality and lower the cost of design for the design of PCB using CAD systems. The best results of PCB designer was achieved by such companies as Altium Designer, Mentor Graphics, National Instruments, Zuken. These companies develop packages for designing PCB, which have some advantages: large elements libraries, conversion of the files, inerrability of them in other CAD-systems, comfortable interface and etc. However, these systems have several disadvantages: the tasks consist only one criteria, ignore criteria of thermal conditions, in several systems there is also no automated placement of the EC on PCB. In this regard, there is a need to improve math and CAD software. The novelty of this work is to use the algorithm, for solving multi-criteria problem for placement EC on PCB.

II. PROBLEM STATEMENT

A. The purpose of the work

The purpose of the work is the development of a modified genetic algorithm of electronics component placement on a PCB, considering criteria of thermal conditions and minimum weighted sum.

B. Problem statement

Statement of the placement problem is to find the location coordinates of EC placement on PCB with the predefined criteria and constraints.

III. CRITERIA AND COONDITIONS FOR LOCATION PROBLEM

A criterion of thermal conditions is proposed for placement of elements. Fitness function for this criteria can be expressed as:

$$f_t = \sum_{i=1}^N \left| \sum_{\substack{j=1 \\ j \neq i}}^N (P_i - P_j) d_{ij} \right| \rightarrow \min \quad (1)$$

where N is number of EC located on PCB; P_i and P_j are the thermal power dissipation i and j EC; d_{ij} - distance between i and j EC.

Using this criterion allows evenly distributing the EC on PCB, which allows improving the quality of PCB.

Furthermore, it is proposed to take into account the criteria of minimum weighted sum. Fitness function for this criteria can be expressed as:

$$f_c = \sum_{i=1}^n \sum_{j=0}^n c_{ij} d_{ij} \rightarrow \min \quad (2)$$

where c_{ij} – number of links between i and j EC.

Using this criterion allows reducing the distance between the maximal connected EC, which simplifies the subsequent tracing and improves the electrical characteristics of the device.

III. GENETIC ALGORITHM FOR PLACEMENT EC ON PCB

A. Genetic algorithm

GA – is a search heuristic that mimics the process of natural evolution.

Key terms for using GA:

- Individual is one potential solution.
- Population is a set of potential solutions.
- Chromosome is code representation of solutions. Gen is cell chromosomes, which can change its value.
- Allele is numeric value of gene.
- Locus is location of a gene in chromosome.
- Fitness function
- Generation - one cycle of the GA, including a procedure for breeding, mutation breeding [3].

B. Genetic algorithm for placement EC on PCB

Genetic algorithm using of the guillotine cutting of material. The plan of cutting is carried a binary tree.

The algorithm of placement EC on PCB using GA include next steps:

Step 0. Input of initial data.

Initial data of the problem are:

1. The number of the EC.
2. The number of links between i and j EC.
3. The thermal power dissipation i and j EC.
4. The number of individuals in a population.
5. The number of generations of evolution.

Step 1. Creation of a new population. While creation of initial population a number individuals set at random. Each gene of chromosome gets its unique value.

Each solution is encoded by two chromosomes: XP1, XP2.

In contracts to the encoding of chromosomes proposed in articles [5], [6], we suggest another method of encoding chromosome XP1.

Chromosome XP1 contains coded information about the laying of the tree leaves, which indicate the order of the placement of EC on PCB: $XP1 = \{g_{li} \mid i = 1, 2, \dots, n\}$. Each gene g_{li} can take any value in the range $[1; n]$.

For example, $n=9$, XP1 (Fig.1): $\langle 4,2,1,3,5,9,6,7,8 \rangle$.

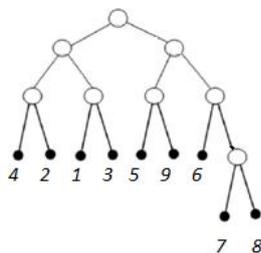


Fig.1. Binary tree for chromosome XP1

The proposed modification of the algorithm can significantly reduce the calculating time to solve the problem.

Chromosome XP2 contains coded information about the type of cut of PCB space.

Chromosome $XP2 = \{g_{2i} \mid i = 1, 2, \dots, n\}$ is the type of the cut (H or V).

The number of the gene XP2 is $n \cdot n - 1$.

The gene value is 0 or 1, with 0 – V – vertical cutting, and 1 – H – horizontal cutting.

For example:

Lets, $n=9$, $XP2 = \langle 0, 1, 0, 0, 1, 1, 0, 1 \rangle$ (Fig. 2).

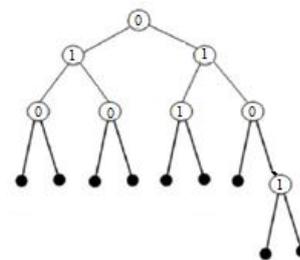


Fig. 2. Binary tree for chromosome XP2

Step 2. Crossover. All individuals of the population are formed in pairs at random. As soon as 2 solution-parents are selected, they are applied to a single-point crossover, which create two new solutions-offspring on their bases. Randomly selects one of the possible break point (Break point is the area between the adjacent bits in the string). Both parental structure are torn into two segments at this point. Then, segments of different parents stick together and produce two offsprings (Fig.3).

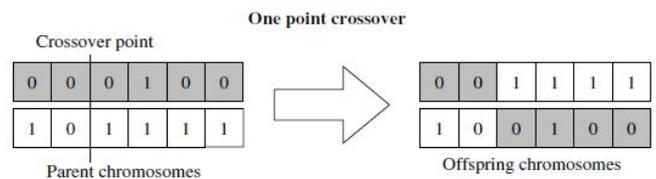


Fig.3. One point crossover

Step 3. Mutation. In case of mutations chromosomes undergoes some accidental modifications. In this work we propose using one-point mutation, where one bit in chromosome selects randomly and changes its value to the opposite one (Fig.4).

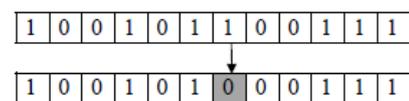


Fig.4. One-point mutation

Step 4. Designer of a binary tree cuts. Knowing the number EC, we can design the binary tree.

The total amount of EC (E_i) – is the top of parent (V_0). If E_i is not equal 2 or 3, E_i will divide by 2. The obtained values are rounded: the first figure in the smallest way, the second figure in the biggest way. The obtained tops (second level) are daughter vertex for V_0 . The first figure will be the first daughter vertex V_1 for V_0 , the second figure will be the second daughter vertex V_2 for V_0 .

If the value of obtained daughter vertex V_1 and V_2 is not equal 2 or 3, they are the top – parents, and they again might be halved. Halving will continue until all of the number of top-parents will be equal to two or three. With each subsequent level, the number of vertices will increase by 2^k , where k – level (Fig.5).

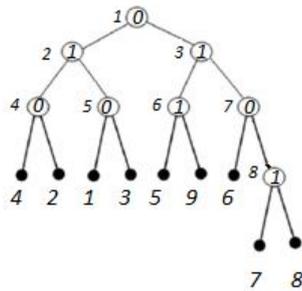


Fig.5. Binary tree cuts

When the number of top-parents is equal to 2, then top-parent correspond to the two binary tree leaves (Fig.6).



Fig.6. The example of the design binary tree cut

When number of top-parents is equal to 3, then top-parent corresponds to another top-parent, which owns two of the leaves and another binary tree leaf (Fig.7).

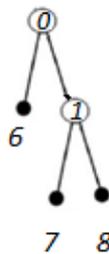


Fig.7. The example of the design of binary tree cut

Step 5. Binary convolution method. On the basis of this information, binary tree designer is carried out by means of serial binary convolution areas on the tree incisions, starting with tree leaves. Each inner top of binary tree corresponds the area, obtained in the result of the binary convolution of a subtree, with the root of the inner top. Consider that cut with number i is cutting the top d_i (area is u_i). In the beginning of convolution each top d_i , which is the leaf of the tree, is corresponding the area s_i with dimension $x_i = a_i, y_i = b_i$, that is equal to module size E_i . For each inner vertex d_i of tree corresponds the area u_i , which is formed by convolution subtree cuts, and has as its root vertex d_i .

Let's consider, that vertexes d_i and d_j are daughter vertex of d_k and the areas u_i и u_j - corresponding d_i and d_j - are determined lower limits of their size (x, y_i), (x_j, y_j). The binary convolution is a fusion of areas u_i and u_j , formation of u_k , dimensioning for u_k and new sizes for u_i and u_j . Let us introduce two infix operators H and V.

The record $u_k = u_i H u_j$, means, that areas u_i and u_j merge horizontally in one area u_k . If $u_k = u_i B u_j$, then the areas u_i and u_j merge vertically.

Designate that $\max(x_1, x_2)$ is maximum value of x_1 and x_2 .

At the confluence of the horizontal:

$$y_k = \max(b_i, b_j);$$

$$x_k = a_i + a_j;$$

y_i and y_j will have the size equal $\max(b_i, b_j)$.

At the confluence of the vertical:

$$y_k = b_i + b_j;$$

$$x_k = \max(a_i, a_j);$$

x_i and x_j will have the size $\max(a_i, a_j)$ [4].

For example:

The module dimensions $\langle 4, 2, 1, 3, 5, 9, 6, 7, 8 \rangle$ are fixed. The area dimensions in accordance with the consistent convolution, will be defined as follows:

$$1. u_8 = 7 H 8;$$

$$y_8 = b_7 + b_8;$$

$$x_8 = \max(a_7; a_8).$$

$$2. u_7 = 6 \vee u_8;$$

$$y_7 = \max(b_6; y_8);$$

$$x_7 = a_6 + x_8) \text{ and etc.}$$

The final plan of allocated modules is presented in figure 8.

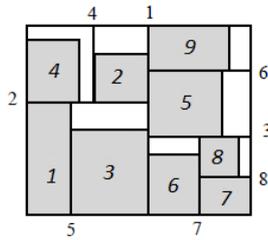


Fig. 8. The final plan of allocated modules

Step 6. Fitness function calculation.

Step 7. Selection. At each step of the evolution, individuals for next iteration are selected with the help of randomly selection operator. Selection allocates more copies of those solutions with higher fitness values and thus imposes the survival-of-the-fittest mechanism on the candidate solutions. The main idea of selection is to prefer better solutions to worse ones.

Step 8. The condition for the completion of the algorithm is the examination of the required number of iterations of the algorithm.

Step 9. Definition of the best individual (solution). The best solution of placement EC on PCB is the solution, which has the least result of fitness function.

Step 10. Result output.

IV. ACKNOWLEDGMENT

To confirm the efficiency of the algorithm for placement EC on PCB using the genetic algorithm of guillotine cutting method a special software was developed.

The study of the efficiency was carried out.

Investigation №1. Test launches of the program with constant initial data and the change in the number of generations of evolution were carried out.

The source and resulting data are presented in table 1.

TABLE I.

The number of the placement EC	The number of individuals in a population	The number of generations of evolution	f_c
50	10	10	666985
50	10	15	652074
50	10	20	587125
50	10	25	562158
50	10	30	478745
50	10	35	436585
50	10	40	395781
50	10	45	325641
50	10	50	284578
50	10	55	256485
50	10	60	256484
50	10	65	256484
50	10	70	256484

The table I is shown that with the increase of the generations of the evolution, the fitness-function decreases or remains unchanged, which testifies the optimal placement of the EC on the PP.

Investigation №2. Made a comparison of an iterative algorithm (IA) with our GA placement of EC on the PCB with the constant initial data.

TABLE II.

The number of the placement EC	f_{cGA}	f_{cIA}
10	5201	10257
20	82114	320458
30	157482	658921
40	304580	845225
50	478745	1036586
60	658148	1203558
70	851482	1698014
80	1201425	2003688
90	1582012	2365247
100	1965214	2875218

The table II is shown, that the values of the fitness-function GA is significantly less than the target function of iterative algorithm. Therefore, GA is more efficient than the standard algorithm.

V. CONCLUSION

In this paper proposed a developed software that implements automated procedures EC on PCB.

An important advantage of this software lies in the possibility taking into account the criteria of thermal conditions, while implementing the project the procedures, that will allow to increase the quality and reliability of the device.

The developed software integrates CAD-system Mentor Graphics Expedition PCB, which significantly increases the ease of use for end users.

In the future, the developed software product will be applied on a number of enterprises of Tatarstan for the design and development of printed circuit boards: "Радиоприбор", "Electronics", etc.

REFERENCES

- [1] *Воронова В.В.* Автоматизация проектирования электронных средств: Учебное пособие. Казань: Изд-во гос. техн. ун-та им. А.Туполева, 2000.-67 с/
- [2] *Родзин С.И.* Гибридные интеллектуальные системы на основе алгоритмов эволюционного программирования // Новости искусственного интеллекта. 2000. №3. С. 159-170.
- [3] *Овчинников В.А., Васильев А.Н., Лебедев В.В.* Проектирование печатных плат: Учебное пособие. 1-е изд. Тверь: ТГТУ, 2005.116 с.
- [4] *Лебедев Б.К., Лебедев В.Б.*, Адаптивная процедура выбора ориентации модулей при планировании СБИС// Известия ЮФУ. Технические науки. Тематический выпуск «Интеллектуальные САПР». – Таганрог: Изд-во ТТИ ЮФУ, 2010, № 7 (108). – 260 с.
- [5] *Лебедев Б.К., Лебедев В.Б.*, Планирование на основе роевого интеллекта и генетической эволюции// Известия ЮФУ. Технические науки. Тематический выпуск «Интеллектуальные САПР». – Таганрог: Изд-во ТТИ ЮФУ, 2009, № 4 (93). – 254 с.