

# **SYRC<sub>o</sub>SE 2011**

Editors:

Alexander Kamkin, Alexander Petrenko,  
Andrey Terekhov

Proceedings of the 5<sup>th</sup> Spring/Summer Young Researchers' Colloquium on  
Software Engineering

Yekaterinburg, May 12-13, 2011

Yekaterinburg  
2011

**Proceedings of the 5<sup>th</sup> Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2011), May 12-13, 2011 – Yekaterinburg, Russia:**

The issue contains the papers presented at the 5<sup>th</sup> Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2011) held in Yekaterinburg, Russia on 12<sup>th</sup> and 13<sup>th</sup> of May, 2011. 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 system programming; static verification and analysis of programs; programming languages, methods and tools; testing of software and hardware systems; automata theory; computer graphics and others.

**Труды 5-ого весеннего/летнего коллоквиума молодых исследователей в области программной инженерии (SYRCoSE 2011), 12-13 мая 2011 г. – Екатеринбург, Россия:**

Сборник содержит статьи, представленные на 5-ом весеннем/летнем коллоквиуме молодых исследователей в области программной инженерии, проводимом в Екатеринбурге 12-13 мая 2011 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: системное программирование; статическая верификация и анализ программ; языки, методы и инструменты программирования; тестирование программных и аппаратных систем; теория автоматов; компьютерная графика и другие.

ISBN 978-5-91474-017-4

# Contents

Foreword.....5

Committees / Referees.....6

## Guest Talk

Crowdsourcing Projects for Research, Education and Better Life  
*R. Yavorskiy*.....8

## Biomolecular Computing

tRNA Computing Units and Programming Languages  
*N. Odincova, V. Popov*.....10

## System Programming

Using Hardware-Assisted Virtualization to Protect Application Address Space Inside Untrusted Environment  
*D. Silakov*.....17

Background Optimization in Full System Binary Translation  
*R. Sokolov, A. Ermolovich*.....25

The ARTCP Header Structure, Computation and Processing in the Network Subsystem of Linux Kernel  
*A. Sivov, V. Sokolov*.....31

## Information Representation, Search and Reasoning

A New Double Sorting-Based Node Splitting Algorithm for R-Tree  
*A. Korotkov*.....36

Fuzzy Matching for Partial XML Merge  
*V. Fedotov*.....42

High-level Data Access Based on Query Rewritings  
*E. Stepalina*.....45

Application of the Functional Programming Tools in the Tasks of Language and Interlanguage Structures Representation  
*P. Ermakov, O. Kozhunova*.....48

## Static Verification and Analysis of Programs

Static Verification Under The Hood: Implementation Details and Improvements of BLAST  
*P. Shved, V. Mutilin, M. Mandrykin*.....54

Detecting C Program Vulnerabilities  
*A. Ermakov, N. Kushik*.....61

Model Checking Approach to the Correctness Proof of Complex Systems  
*M. Alekseeva, E. Dashkova*.....65

## **Programming Languages, Methods and Tools**

Thorn Language: a Flexible Tool for Code Generation <i>Y. Okulovsky</i> .....	68
One Approach to Aspect-Oriented Programming Implementation for the C Programming Language <i>E. Novikov</i> .....	74
Component-Based Software Engineering and Runtime Type Definition <i>A. Shakurov</i> .....	82

## **Software Engineering Education**

Educational Tests in “Programming” Academic Subject Development <i>O. Maksimenkova, V. Podbelskiy</i> .....	88
--	----

## **Automata Theory**

The Parallel Composition of Timed Finite State Machines <i>O. Kondratyeva, M. Gromov</i> .....	94
Separating Non-Deterministic Finite State Machines with Time-outs <i>R. Galimullin, N. Shabaldina</i> .....	100

## **Testing of Software and Hardware Systems**

Model Based Conformance Testing for Extensible Internet Protocols <i>N. Pakulin, A. Tugaenko</i> .....	105
Developing Test Systems for Multi-Modules Hardware Designs <i>M. Chupilko</i> .....	111

## **Application-Specific Methods and Tools**

Programming for Modular Reconfigurable Robots <i>A. Gorbenko, V. Popov</i> .....	117
Towards a Real-Time Simulation Environment on the Edge of Current Trends <i>E. Chemeritskiy, K. Savenkov</i> .....	128

## **Computer Graphics and Vision**

The Problem of Placement of Visual Landmarks <i>A. Gorbenko, M. Mornev, V Popov</i> .....	134
Hand Recognition in Live-Streaming Video <i>M. Belov</i> .....	142
3D-Illusion Constructor <i>M. Rovkin, E. Yel'chugin, M. Filatova</i> .....	145

## Foreword

Dear participants, we are glad to welcome you on the 5<sup>th</sup> Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). This year we have the pleasure of holding SYRCoSE in Yekaterinburg, the main industrial and cultural center of the Urals Federal District. The colloquium is hosted by the Ural State University (USU), one of the most prestigious universities in Russia. The event is organized by Institute for System Programming of RAS (ISPRAS) and Saint-Petersburg State University (SPSU) jointly with USU and SKB Kontur.

Program Committee has selected 24 papers that cover different topics of software engineering and computer science. Each submitted paper has been reviewed independently by two or three referees. Participants of SYRCoSE 2011 represent well-known universities, research institutes and IT companies such as Institute of Informatics Problems of RAS (IPI RAN), Intel, ISPRAS, MCST, Moscow State University, National Research Nuclear University "MEPhI", State University – Higher School of Economics, Tomsk State University, USU and Yaroslavl Demidov State University.

We would like to thank all the participants of SYRCoSE 2011 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, Petr Skobelev (SEC "Knowledge Genesis") and Shihong Huang (Florida Atlantic University). We would also like to thank our sponsors, Russian Foundation for Basic Research (grant 11-07-06013-r) and Microsoft Research. Finally, our special thanks to Irina Voychitskaya (SKB Kontur), Maria Rudnichenko (SKB Kontur) and Tatyana Perevalova (USU) for their invaluable help in organizing the colloquium in Yekaterinburg.


Sincerely yours

Alexander Kamkin, Alexander Petrenko, Andrey Terekhov  
May 2011

# Committees

## Program Committee Chairs

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

 Andrey N. Terekhov – Russia  
*Saint-Petersburg State University*

## Program Committee

 Habib Abdulrab – France  
*National Institute of Applied Sciences, INSA-Rouen*

 Igor V. Mashechkin – Russia  
*Moscow State University*


 Sergey M. Avdoshin – Russia  
*Higher School of Economics*

 Alexander S. Mikhaylov – Russia  
*National Research Nuclear University "MEPHI"*

 Eduard A. Babkin – Russia  
*Higher School of Economics*

 Valery A. Nepomniaschy – Russia  
*Ershov Institute of Informatics Systems*

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

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

 Efim M. Grinkrug – Russia  
*Higher School of Economics*

 Vladimir Yu. Popov – Russia  
*Ural State University*


 Maxim L. Gromov – Russia  
*Tomsk State University*


 Ruslan L. Smelyansky – Russia  
*Moscow State University*

 Vladimir I. Hahanov – Ukraine  
*Kharkov National University of Radioelectronics*

 Valeriy A. Sokolov – Russia  
*Yaroslavl Demidov State University*


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

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

 Oleg R. Kozyrev – Russia  
*Higher School of Economics*

 Mikhail V. Volkov – Russia  
*Ural State University*

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

 Rostislav E. Yavorsky – Russia  
*Microsoft*

 Yury S. Lukach – Russia  
*Ural State University*

 Nina V. Yevtushenko – Russia  
*Tomsk State University*

 Tiziana Margaria – Germany  
*University of Potsdam*

 Vladimir A. Zakharov – Russia  
*Moscow State University*

## Organizing Committee Chairs

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

 Irina A. Voychitskaya  
*SKB Kontur*

## Organizing Committee

 Yury S. Lukach – Russia  
*Ural State University*

 Maria A. Rudnichenko  
*SKB Kontur*

 Tatyana V. Perevalova  
*Ural State University*

 Mikhail V. Volkov  
*Ural State University*

 Vladimir Yu. Popov – Russia  
*Ural State University*

## Referees

Vladimir BASHKIN

Victor GERGEL

Efim GRINKRUG

Maxim GROMOV

Vladimir HAHANOV

Alexander KAMKIN

Vsevolod KOTLYAROV

Victor KULIAMIN

Natalia KUSHIK

Egor KUZMIN

Alexander LETICHEVSKY

Tiziana MARGARIA

Artem MELENTYEV

Alexander MIKHAYLOV

Valery NEPOMNIASCHY

Dmitry PAVLENKO

Alexander PETRENKO

Ivan PILETSKI

Vladimir POPOV

Svetlana PROKOPENKO

Alexey PROMSKY

Natalia SHABALDINA

Valery SOKOLOV

Maria VETROVA

Mikhail VOLKOV

Rostislav YAVORSKY

Nina YEVTUSHENKO

Vladimir ZAKHAROV

# Crowdsourcing Projects for Research, Education and Better Life

Extended abstract

Rostislav E. Yavorskiy  
Vice President, Models and Algorithms  
Witology, <http://witology.com>  
Office 215, Building 3, Kapranova Per.  
Moscow, 123242, Russia

**Abstract.** This paper provides a brief introduction into two open source projects in the area of Computer Science and Software Engineering in Russia

*Crowdsourcing, open source projects, software engineering education, computer science, DOM API testing, wild fires monitoring*

## I. INTRODUCTION

According to Wikipedia [1] “Crowdsourcing is the act of outsourcing tasks, traditionally performed by an employee or contractor, to an undefined, large group of people or community (a “crowd”), through an open call.” This paper describes two initiatives of this format, where a wide community of students and young researchers has been invited to participate in solving a big and important problem. The both projects are still active, so another goal of this paper is to attract new activists to join.

## II. DOM API TESTING FOR CONFORMANCE TO W3C STANDARD

Document Object Model Application Programming Interface (DOM API) standard specifies an interface for accessing and manipulating documents programmatically in Web browsers, see [2]. W3C group provides conformance tests for DOM API; statistics on test case numbers is presented in [5], see table 3. We just mention here that the standard specifies 946 methods and attributes, and some modules are not covered by the test suite at all.

A very rough estimate shows that deep and detailed analysis and test development for a DOM element may require up to one working week, so development of a full test suite would require approximately 20 men-years.

The goal of our project is to involve wide community into the work of creating the complete test suite, see [3].

### A. Research

Creating test suites with a good coverage metrics is a challenging research task. One may start with [4-5] and then follow the references.

### B. Education

This project could be used as a good topic for course paper, diploma or even PhD thesis by students of Computer Science and Software Engineering departments of universities.

### C. Better life

Taking into account the role of Internet in our life, there is no need to say much on justifying the importance of robustness and interoperability of Web browsers.

## III. USING SATELLITE IMAGES FOR MONITORING OF WILD FIRES

Wild fire is a regular phenomenon, which causes extensive damage both to property and human life. Satellite images provide a good tool for studying, analyzing, and predicting wild fires. A huge amount of satellite data is available online, see e.g. MODIS page [6].

Our project [7] is aimed at involving wide community of students and researchers into the work of analyzing the satellite images and developing tools and algorithms for better monitoring and predicting wild fires.

### A. Research

There are four rather independent research areas, related to this project.

- *Scientific Databases.* See e.g. project on “Environmental Scenario Search Engine” [8].
- *Image Recognition.* See Open CV project for more [9-10].
- *Scientific Data Management and Visualization.* See e.g. Scientific Data Set project [11].
- *Domain specific research.* For example, models and critical factors for wild fires appearance and spreading.

### B. Education

Similarly, this project provides numerous interesting examples to be used at the relevant courses in universities.

---

The work described in this paper is partially sponsored by Microsoft Rus and Microsoft Research



### C. *Better life*

The importance of this project is obvious.

#### REFERENCES

- [1] Wikipedia, The Free Encyclopedia, Crowdsourcing <http://en.wikipedia.org/wiki/Crowdsourcing>
- [2] Document Object Model, <http://www.w3.org/DOM/>
- [3] DOM API Testing for conformance against W3C standard, <http://domapitesting.codeplex.com>
- [4] DOM API Contracts and Test Suite Development Using Code Contracts and Pex. Research Report. Institute for System Programming, Russian Academy of Sciences (ISP RAS) by order of Microsoft Research, November 2009. See <http://domapitesting.codeplex.com/documentation>
- [5] Test Development for DOM Support in Internet Browsers. Research report. Institute for System Programming, Russian Academy of Sciences (ISP RAS) by order of Microsoft Research, June 2010. See <http://domapitesting.codeplex.com/documentation>
- [6] MODIS, Moderate Resolution Imaging Spectroradiometer, <http://modis.gsfc.nasa.gov/>
- [7] Monitoring of Wild Fires Project (in Russian) <http://gis-lab.info/projects/fires.html>
- [8] Open-source project Environmental Scenario Search Engine, <http://esse.wdcb.ru/>
- [9] И. Лысенков, Распознавание сгоревших территорий с помощью деревьев решений и OpenCV, <http://gis-lab.info/qa/burnedarea-opencv.html>
- [10] Open Source Computer Vision library, <http://opencv.willowgarage.com/wiki/>
- [11] SDS: Scientific DataSet library and tools, <http://sds.codeplex.com/>

# tRNA Computing Units and Programming Languages

Natalya Odincova  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: odincova.antalya@gmail.com

Vladimir Popov  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: Vladimir.Popov@usu.ru

**Abstract**—In this paper we consider some new computing units for DNA-based computers. Construction of such unit essentially based on properties of tRNA. Therefore, we call them as tRNA computing units.

## I. INTRODUCTION

In the recent years several new ideas have been developed to use non electronic natural phenomena for real, efficient computation. In classical electronic-based computations the information is stored and modified bitwise by electric and electromagnetic means. It is typical for this kind of computations that the number of steps performed per time unit is huge but the number of processors running in parallel is small. The main objective for the new approaches mentioned above is not to speed up the number of steps per time unit but to increase the degree of parallelism considerably. In 1985 D. Deutsch [1] proposed computers using quantum-physical effects to store and modify information. The quasi-probabilistic physical effect of quantum parallelism and mutual dependences of between all bits (coherence effects) allow to construct quantum algorithms that solve certain problems faster than any known probabilistic algorithm. In [1] Quantum Turing Machines are introduced as a theoretical model of such a kind of computation. In [2] quantum machine algorithms for the discrete logarithm and for integer factoring are given which run in polynomial time. In 1994 different approaches came up that used biological properties of DNA strings to store and modify information. The general idea is to use a large number of DNA strings as processors which compute in parallel. In [3] P. Pudlák introduced Genetic Turing Machines that are probabilistic machines which can simulate the evolution of a population of strings using two special operators controlling the inheritance and the survival of strings. In this model on each of the randomly chosen paths one string is processed. Also in 1994, L. Adleman [4] used biological experiments with DNA strings to solve some particular instances of the directed hamiltonian path problem which is considered to be intractable because of its **NP**-completeness. In [5] – [7] R. Lipton showed how to extend this idea to solve any problem and discussed the practical relevance of this approach. He defined a model of biological computing that has, besides the

classical means, the ability of manipulating large collections of DNA strings. Performing one of the special operations on a test tube means some simple manipulation of each of the strings in the test tube. In that way each DNA string corresponds to a piece of information, and all these pieces can be modified in parallel. At current DNA manipulation technology levels, DNA computing provides no advantage over electronic computers, for example, when encoding the computing task with DNA molecule in Adlemans directed hamiltonian path problem, if the  $n$  is equal to 100, the amount of DNA required would be larger than the weight of the earth. There is not enough room for improvement on algorithm to make the number of DNA molecules practically small. At this stage, some people began to worry about the directions of DNA computing study. However, in other sub-fields of DNA computing, great progress has been made. There are currently several research disciplines driving towards the creation and use of DNA nanostructures for both biological and non-biological applications. These converging areas are:

- the miniaturization of biosensors and biochips into the nanometer scale regime;
- the fabrication of nano-scale objects that can be placed in intracellular locations for monitoring and modifying cell function;
- the replacement of silicon devices with nano-scale molecular-based computational systems, and the application of biopolymers in the formation of novel nano-structured materials with unique optical and selective transport properties.

DNA computing employs DNA molecule as a main resource to fulfill computing tasks. However, the concept of primary DNA computing unit keeps obscure. It is recently realized that there are multiple forms of basic DNA computing units. Adleman uses short oligonucleotides to encode mathematical problems. The computing process is mainly performed in the form of hybridization. Ligation and other molecular manipulation steps are used for output abstraction. The correct answer is hidden in a vast amount of different hybridization results. Rothmund proposed a Turing machinelike DNA computing unit [8]. In [9] – [11] published another study in which

an autonomous programmable DNA automaton is created. In particular, in [9] – [11] for DNA automaton used a double-stranded DNA as input, endonuclease and DNA ligase as main hardware, transition molecules as software, thus creating a two-state molecular finite automaton with a two-symbol input, eight transition rules and 765 syntactically distinct programs. DNA self-assembly has become one of the most important directions for DNA computing [12] – [19]. Because of its universal computing capability, DNA assembly provides another avenue for universal DNA computer development. DNA computing by self-assembly is basically a tiling process, and the tile types can vary a lot. The tiles can be formed with several singlestranded oligos, and each tile can have different sticky DNA ends for a number of combinations with other same or different tiles. The tiling can be designed in a twodimensional or three-dimensional way, and the scale for tiling should also be able to control. DNA assembly can be completely programmed, though molecular biology experiments are still a bottle-neck for large scale assembly. In [19] authors brought a new landscape for this avenue. Combinatorial cellular automata also used in designing any tiling shapes. Besides, the natural affinity of DNA to bind with proteins, some types of small molecules, even metal atoms, makes it possible that assembled DNA can work as an inherent or transient matrix for novel computing devices. In [20] – [23] published a study in ribozyme unit research area. Ribozyme is a piece of nucleic acid fragment with unique three-dimensional structure that has an enzymatic ability to cut specific complementary oligos as substrate. If another oligo binds with the ribozyme and prevents it from forming enzymatic conformation, the ribozyme stays in an inactive form. In [20] – [23] founded ribozymes that can be easily manipulated as logical gates. Thus such ribozyme can mimic conventional electronic computing devices and theoretically develop universal DNA computing system. Ribozymes can work as automaton, though for the time being ribozyme or deoxyribozyme automaton is still in its infancy. Ribozymebased DNA computing unit may be extremely useful in designing logical computing devices in the future, for example, single-molecule logical gate. In [24], [25] also trying to employ ribozyme-based DNA computing as a potential vehicle for *in vivo* DNA computing. Instead of making ribozymes into logical gates or automata, in [24], [25] ribozymes used to build simple automata that may be easier for *in vivo* usage. Membrane computing [26], [27] can be regarded as a unique biological computing system. A cell is the basic unit for membrane computing system. This unit is not a DNA computing unit. However, membrane system provides another sort of self-assembly tile, and each such unit can hold DNA in it and may be able to translocate DNA molecules between each unit in the future, so we would like to treat such unit as a special DNA computing unit. It might be also called cell computing, a natural distributed architecture of a computing unit where any other DNA computing unit processes might be embedded. Since no kind of artificial membrane computing systems has been tested in the form of biochemical or physical biochemical experiments, it is likely that the natural cells may

be firstly tried by cell molecular biology manipulations. So some *in vivo* DNA computing technology may be needed to develop beforehand.

In this paper we consider some new DNA computing units. Construction of such unit essentially based on properties of tRNA. Therefore, we call them as tRNA computing units.

## II. tRNA COMPUTING UNITS

Transfer RNA (tRNA) is RNA that transfers a specific active amino acid to a growing polypeptide chain at the ribosomal site of protein synthesis during translation. tRNA has a 3' terminal site for amino acid attachment. This covalent linkage is catalyzed by an aminoacyl tRNA synthetase. It also contains a three base region called the anticodon that can base pair to the corresponding three base codon region on mRNA. Each type of tRNA molecule can be attached to only one type of amino acid, but because the genetic code contains multiple codons that specify the same amino acid, tRNA molecules bearing different anticodons may also carry the same amino acid. An anticodon [28] is a unit made up of three nucleotides that correspond to the three bases of the codon on the mRNA. Each tRNA contains a specific anticodon triplet sequence that can base-pair to one or more codons for an amino acid. To provide a one-to-one correspondence between tRNA molecules and codons that specify amino acids, 61 types of tRNA molecules would be required per cell. However, many cells contain fewer than 61 types of tRNAs because the wobble base is capable of binding to several, though not necessarily all, of the codons that specify a particular amino acid. A minimum of 31 tRNA are required to translate, unambiguously, all 61 sense codons of the standard genetic code [29].

The main function of tRNA is to recognize a fragment of single-stranded DNA molecule which consists of three nucleotides. As a result of such action is established a correspondence between the triplet of nucleotides of DNA nucleotides and a triple contact element of the tRNA molecule. *In vivo* tRNA molecule used for the amino acids synthesis. However, at least *in vitro* using special enzymes, we can stop the natural process of protein synthesis at the stage of reading nucleotide triplets of the DNA molecule and start the process of reading information from the tRNA molecules. As a result, we obtain a new DNA molecule. In the classical model of tRNA function we do not get anything interesting. In view of one-to-one correspondence between triples of DNA and tRNA we simply obtain a copy or some subsequence of the original DNA molecule. More precisely, *in vitro* we can produce any of following operations. Let

$$F[P](x) = y$$

where  $F[P](x)$  is a function with a parameter  $P$  of the variable  $x$  defined as follows:

$$x = x[1]z[1]x[2]z[2] \dots x[n]z[n]x[n+1],$$

$$y = z[1]z[2] \dots z[n],$$

$$\begin{aligned}
z[i] \in P \subseteq \mathcal{P} = \{ &UUU, UUC, UUA, \\
&UUG, UCU, UCC, UCA, UCG, \\
&UAU, UAC, UGU, UGC, UGG, \\
&CUU, CUC, CUA, CUG, CCU, \\
&CCC, CCA, CCG, CAU, CAC, \\
&CAA, CAG, CGU, CGC, CGA, \\
&CGG, AUU, AUC, AUA, AUG, \\
&ACU, ACC, ACA, ACG, AAU, \\
&AAC, AAA, AAG, AGU, AGC, \\
&AGA, AGG, GUU, GUC, GUA, \\
&GUG, GCU, GCC, GCA, GCG, \\
&GAU, GAC, GAA, GAG, GGU, \\
&GGC, GGA, GGG\}^+, \\
1 \leq i \leq n, \\
x[j] \in \{A, U, C, G\}^*, \\
1 \leq j \leq n + 1.
\end{aligned}$$

Note that *in vitro* the length of each word  $x[j]$  depends on the specific experimental conditions and the presence in this words subwords from

$$\{UAA, UGA, UAG\}.$$

In general case we can suppose that the length of  $x[j]$  is an arbitrary number. *In vivo*

$$x[j] \in \{UAA, UGA, UAG\}^*.$$

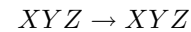
So, we obtain some set of tRNA computing units each of which is given by some operation  $F[P](x)$ . We call them as classical tRNA computing units.

Complete sets of tRNAs from one organism, including at least one isoacceptor species for each of the twenty amino acids, are known for several eubacteria (*Mycoplasma capricolum*, *Bacillus subtilis*, *Escherichia coli*), yeast (*Saccharomyces cerevisiae*) and chloroplasts (*Euglena gracilis*, *Marchantia polymorpha*, *Nicotiana tabacum*) or mitochondria (*Torulopsis glabra*, *ratus ratus*). The number of genes for a particular isoaccepting tRNA varies depending on the organism. Although these genes might have the same primary structure, it is more common that isoacceptor tRNAs feature the same anticodon but slightly differing sequences. In yeast, for example, the two tRNA<sub>GAA</sub><sup>phe</sup> [31] and the two tRNA<sub>IGU</sub><sup>thr</sup> [32] are identical except two nucleotides. Compensatory mutations frequently occur in the case when the difference between two isoacceptors is located in a stem. Again in yeast tRNA<sup>phe</sup>, an A-U base pair in the amino acid acceptor stem is exchanged for a G-C pair. The same replacement is found in yeast tRNA.

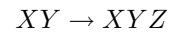
Transfer RNA is the most extensively modified nucleic acid in the cell. Modified nucleotides are contained in tRNAs from

all three phylogenetic domains (*archaea*, *bacteria*, *eucarya* [33], [34]). The modifications are not introduced during transcription, but are formed after the synthesis of the polynucleotide chain, serving for an improvement of the specificity and efficiency of tRNA biological functions. To date, more than eighty modified residues have been discovered and their chemical structures revealed [35]. Modified nucleotides are located at 61 different positions in tRNAs, mainly in loop regions. A large variety is present in the anticodon area, especially in the first position of the anticodon (position 34), and one base 3' to the anticodon (position 37). Apart from one exception (*archaeosine* at position 15 in archael tRNAs [36], all hypermodified residues are found in this region. Minor modifications like methylated or thiolated derivatives are usually situated outside the anticodon, with only one or two kinds of modified nucleotides present at each position. Some are common to almost all species, such as Dihydrouridine in D loops and Ribothymine in T loops, whereas others are characteristic of specific tRNAs. Examples are found in the hypermodified wybutosine residue (a guanosine derivative) at position 37 in almost all eukaryotic tRNA<sup>phe</sup> (except that from *Bombyx mori* and *Drosophila melanogaster*) and queuosine (another complicated post-transcriptional modification of guanosine) at the first anticodon position of certain tRNAs specific for tyr, his, asn and asp from eubacteria and eukaryotes. In both domains, modification takes place at different stages during the processing of precursor tRNA, depending strongly on the concentration of the substrate as well as on both the amount and the activity of tRNA-modifying enzymes. Several studies have been carried out on precursor tRNA<sup>tyr</sup>. The biosynthesis in *Xenopus laevis oocytes* initiated by injection of the yeast tRNA<sup>tyr</sup> gene into either the nucleus or the cytoplasm revealed that most base modifications occur in a sequential fashion in the nucleus before splicing [37], [38].

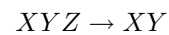
In many cases, the third nucleotide of the contact element of mutant tRNA is not functional. Non-functionality of third nucleotide of the contact element is connected with various mutations that lead to changes in the secondary and the tertiary structures of tRNA. It should be noted that these changes are stable. Note also that these mutations are quite common. In particular, the synthesis of some vital proteins of *Homo Sapiens* is only possible with the assistance of some mutant tRNA. The classical transformation



for some such mutant tRNAs can be represented in form



in case when a third nucleotide of tRNA contact element does not functional for the original DNA and for some other tRNAs can be represented in form



in case when a third nucleotide of tRNA contact element does not functional for the new DNA [30]. So, we obtain the

following set of relations:

$$\begin{aligned}
\mathcal{S} = \{ & UUU = UU, \\
& UUC = UU, UUA = UU, UUG = UU, \\
& UCU = UC, UCC = UC, UCA = UC, \\
& UCG = UC, UAU = UA, UAC = UA, \\
& UGU = UG, UGC = UG, UGG = UG, \\
& CUU = CU, CUC = CU, CUA = CU, \\
& CUG = CU, CCU = CC, CCC = CC, \\
& CCA = CC, CCG = CC, CAU = CA, \\
& CAC = CA, CAA = CA, CAG = CA, \\
& CGU = CG, CGC = CG, CGA = CG, \\
& CGG = CG, AUU = AU, AUC = AU, \\
& AUA = AU, AUG = AU, ACU = AC, \\
& ACC = AC, ACA = AC, ACG = AC, \\
& AAU = AA, AAC = AA, AAA = AA, \\
& AAG = AA, AGU = AG, AGC = AG, \\
& AGA = AG, AGG = AG, GUU = GU, \\
& GUC = GU, GUA = GU, GUG = GU, \\
& GCU = GC, GCC = GC, GCA = GC, \\
& GCG = GC, GAU = GA, GAC = GA, \\
& GAA = GA, GAG = GA, GGU = GG, \\
& GGC = GG, GGA = GG, GGG = GG \}
\end{aligned}$$

where

$$XYZ = XY$$

denotes the pair

$$XYZ \rightarrow XY,$$

$$XY \rightarrow XYZ.$$

We can produce any of following operations. Let

$$G[S, P](x) = y$$

where  $G[S, P](x)$  is a function with parameters  $S$  and  $P$  of the variable  $x$  defined as follows:

$$\begin{aligned}
x &= x[1]x[2] \dots x[n], \\
y &= y[1]y[2] \dots y[n], \\
y[i] &= \begin{cases} z, & x[i] \rightarrow z \in S \subseteq \mathcal{S} \\ x[i], & y[i] \in P \subseteq \mathcal{P} \\ \text{empty word} & \end{cases}
\end{aligned}$$

So, we obtain some set of tRNA computing units each of which is given by some operation  $G[S, P](x)$ . We call them as mutational tRNA computing units.

The frequent occurrence of non-canonical G-U base pairs [39] is a noticeable feature of stem regions. Since their first discovery in [40], other possible non-canonical pairs (for example A-A, C-C, C-U, G-A, U-U, U-Y) have been detected in the stems of various tRNAs [41]. G-U pairs, however, occur with the highest frequency. As to stems, a frequently occurring length can be attributed to loops as well. Anticodon and T loops contain seven nucleotides, whereas D loops and variable regions are areas of various lengths. An important discovery regarding the primary structure was made in the early 1970s. Certain positions in tRNAs are occupied by invariant or semi-invariant nucleotides.

Insights concerning characteristic behaviour of natural tRNA molecules were subsequently applied to the design of artificial tRNA molecules [30]. Using the complete sets of identity elements of some E. coli tRNAs as sequence constraints in inverse folding, a large amount of thermodynamically very stable sequences was obtained and subsequently sorted out due to inefficient folding behaviour.

Genes of interest can be selectively metallized via the incorporation of modified triphosphates [42]. These triphosphates bear functions that can be further derivatized with aldehyde groups via the use of click chemistry. Treatment of the aldehyde-labeled gene mixture with the Tollens reagent, followed by a development process, results in the selective metallization of the gene of interest in the presence of natural DNA strands.

In [43] reported a simple solution based method for the gold (Au) metallization of DNA resulting in a Au nanowire network. Advantage of solution based approach is that it allows the removal of excess gold ( $Au^{+3}$ ) ions by extraction with tetraoctylammonium bromide (TOAB) in order to avoid non specific metallization. Further it has been shown that Au metallized DNA obtained in aqueous phase can be transferred to organic phase using hexadecyl aniline (HDA). Au metallized DNA has potential application in nanoscale devices.

Also a number of small organic ribonucleases have been synthesized with rigid polycationic structures containing an aromatic framework with two residues of bis-quaternary salts of 1,4-diazabicyclo[2.2.2]octane (DABCO) bearing various substituents [44]. The compounds carrying positively charged groups connected via rigid linker are expected to bend the sugar-phosphate backbone and can stimulate the intramolecular phosphoester transfer reaction.

Since we can use artificial nucleotides and artificial tRNA molecules, we can consider artificial tRNA computing units. In this case we consider some alphabet  $\Sigma$  and the set of relations

$$\begin{aligned}
\mathcal{Q} = \{ & X_1Y_1Z_1 \rightarrow X_1Y_1Z_1, \\
& X_2Y_2 \rightarrow X_2Y_2Z_2, \\
& X_3Y_3Z_3 \rightarrow X_3Y_3 \mid \\
& X_i, Y_i, Z_i \in \Sigma, 1 \leq i \leq 3 \}.
\end{aligned}$$

We can define following operations. Let

$$H[Q](x) = y$$

where  $H[Q](x)$  is a function with a parameters  $Q$  of the variable  $x$  defined as follows:

$$\begin{aligned} x &= x[1]x[2] \dots x[n], \\ y &= y[1]y[2] \dots y[n], \\ y[i] &= \begin{cases} z, x[i] \rightarrow z \in Q \subseteq \mathcal{Q} \\ \text{empty word} \end{cases} \end{aligned}$$

We obtain the set of tRNA computing units each of which is given by some operation  $H[Q](x)$ . We call them as artificial tRNA computing units.

### III. tRNA PROGRAMMING LANGUAGES

Let  $k \geq 0$  and  $m \geq 1$  be variables for natural numbers, let  $a, b \in \{0, 1\}$ , let  $x$  be a word variable and let  $T, T_1$  and  $T_2$  be set variables. Let  $I(x) \in \{0, 1\}^*$  be the contents of the word variable  $x$ , and let  $I(T) \subseteq \{0, 1\}^*$  be the contents of the set variable  $T$  in a given moment. We define the cut operation  $\setminus$  by  $\setminus av = v$  and  $\setminus \Delta = \Delta$  where  $\Delta$  is the empty word. Different types of DNA-computers use the following instructions with set operations and conditions with set tests [45].

$$\begin{aligned} T &= T_1 \cup T_2, \\ I(T_1) \cup I(T_2); \end{aligned}$$

$$\begin{aligned} T &= In(k), \\ \{0, 1\}^k; \end{aligned}$$

$$\begin{aligned} T &= T_1 \cdot T_2, \\ I(T_1) \cdot I(T_2); \end{aligned}$$

$$\begin{aligned} T &= \setminus T_1, \\ \{\setminus z \mid z \in I(T_1)\}; \end{aligned}$$

$$\begin{aligned} T &= Sw(T_1), \\ \{y \mid \exists v \exists w (vyw \in I(T_1))\}; \end{aligned}$$

$$\begin{aligned} T &= a \cdot T_1, \\ \{a\} \cdot I(T_1); \end{aligned}$$

$$\begin{aligned} T &= Eq(T_1 \cdot m \cdot a), \\ \{vaw \mid (v0w \in I(T_1) \vee v1w \in I(T_1)) \wedge |v| = m - 1\}; \end{aligned}$$

$$\begin{aligned} T &= Bs(T_1 \cdot m \cdot a \cdot b), \\ \{vaw \mid v0w \in I(T_1) \wedge |v| = m - 1\} \cup \\ \{vbw \mid v1w \in I(T_1) \wedge |v| = m - 1\}; \end{aligned}$$

$$\begin{aligned} T &= Bx(T_1 \cdot m \cdot a), \\ I(T_1) \cap (\{0, 1\}^{m-1} a \{0, 1\}^*); \end{aligned}$$

$$\begin{aligned} T &= Br(T_1 \cdot m \cdot a \cdot x), \\ \{vI(x)w \mid vaw \in I(T_1) \wedge |v| = m - 1\}; \end{aligned}$$

$$\begin{aligned} T &= Bl(T_1 \cdot m \cdot a \cdot b), \\ \{vbw \mid vaw \in I(T_1) \wedge |v| = m - 1\}; \end{aligned}$$

$$\begin{aligned} x &\in T, \\ I(x) &\in I(T); \end{aligned}$$

$$\begin{aligned} T &= \emptyset, \\ I(T) &= \emptyset, \end{aligned}$$

$$\begin{aligned} T_1 &\subseteq T_2, \\ I(T_1) &\subseteq I(T_2). \end{aligned}$$

We can use our computing units independently or add them to this computing units. Depending on experimental conditions using the same computing units we can obtain essentially different programming languages. For example, if we allow unrestricted appliance of operations  $F[P](x)$  and  $G[S, P](x)$ , then we can consider the following semigroup as a model of computations:

$$\langle A, U, C, G \mid S \rangle.$$

Note that

$$\begin{aligned} UAA &= UACA = UAC = UA, \\ UGA &= UGGA = UGG = UG, \\ UAG &= UACG = UAC = UA. \end{aligned}$$

Therefore,

$$\begin{aligned} \langle A, U, C, G \mid S \rangle &= \\ \{U, A, C, G, \\ UU, UA, UC, UG, \\ AU, AA, AC, AG, \\ CU, CA, CC, CG, \\ GU, GA, GC, GG\}. \end{aligned}$$

From other hand, using restricted appliance of operations  $F[P](x)$  and  $G[S, P](x)$ , we can easily obtain a semigroup with undecidable word problem.

Note that for tRNA programming languages we have only set and string variables and constants. This is a characteristic feature of all programming languages for DNA computing. In the case of DNA computing, we have significant difficulties with numerical operations and numbers themselves. But “difficult” does not mean “impossible”. For example, suppose that we have a binary register

$$a_1 a_2 a_3 a_4$$

where  $a_1, a_2, a_3, a_4 \in \{0, 1\}$ . Assume that we want to define some bit operations. We can emulate this binary register the following word:

$$GGA_1 GGA_2 GGA_3 GGA_4$$

where

$$A_i = \begin{cases} G, & a_i = 0 \\ C, & a_i = 1 \end{cases}$$

Let

$$G_{0,1,1}[S_1, P] : GGCx[1]x[2]x[3] \rightarrow GGx[1]x[2]x[3],$$

$$G_{0,1,2}[S_2, P] : GGx[1]x[2]x[3] \rightarrow GGGx[1]x[2]x[3],$$

$$G_{0,2,1}[S_1, P] : x[1]GGCx[2]x[3] \rightarrow x[1]GGx[2]x[3],$$

$$G_{0,2,2}[S_2, P] : x[1]GGx[2]x[3] \rightarrow x[1]GGGx[2]x[3],$$

$$G_{0,3,1}[S_1, P] : x[1]x[2]GGCx[3] \rightarrow x[1]x[2]GGx[3],$$

$$G_{0,3,2}[S_2, P] : x[1]x[2]GGx[3] \rightarrow x[1]x[2]GGGx[3],$$

$$G_{0,4,1}[S_1, P] : x[1]x[2]x[3]GGC \rightarrow x[1]x[2]x[3]GG,$$

$$G_{0,4,2}[S_2, P] : x[1]x[2]x[3]GG \rightarrow x[1]x[2]x[3]GGG,$$

$$G_{1,1,1}[S_3, P] : GGGx[1]x[2]x[3] \rightarrow GGx[1]x[2]x[3],$$

$$G_{1,1,2}[S_4, P] : GGx[1]x[2]x[3] \rightarrow GGCx[1]x[2]x[3],$$

$$G_{1,2,1}[S_3, P] : x[1]GGGx[2]x[3] \rightarrow x[1]GGx[2]x[3],$$

$$G_{1,2,2}[S_4, P] : x[1]GGx[2]x[3] \rightarrow x[1]GGCx[2]x[3],$$

$$G_{1,3,1}[S_3, P] : x[1]x[2]GGGx[3] \rightarrow x[1]x[2]GGx[3],$$

$$G_{1,3,2}[S_4, P] : x[1]x[2]GGx[3] \rightarrow x[1]x[2]GGCx[3],$$

$$G_{1,4,1}[S_3, P] : x[1]x[2]x[3]GGG \rightarrow x[1]x[2]x[3]GG,$$

$$G_{1,4,2}[S_4, P] : x[1]x[2]x[3]GG \rightarrow x[1]x[2]x[3]GGC,$$

$$x[i] \in P = \{GGG, GGC\},$$

$$S_1 = \{GGC \rightarrow GG\}, S_2 = \{GG \rightarrow GGC\},$$

$$S_3 = \{GGG \rightarrow GG\}, S_4 = \{GG \rightarrow GGC\}.$$

Let

$$B_{i,j}(x) \rightleftharpoons G_{i,j,2}(G_{i,j,1}(x)).$$

It is easy to check that using  $B_{i,j}$  we can obtain arbitrary bit operations.

## IV. CONCLUSION

In this paper we consider some new computing units which can be used in different programming languages for DNA-based computers.

As the main direction of further research we can mention the rigorous formalization and classification of programming languages based on tRNA computing units and the study of computational power of such programming languages.

## ACKNOWLEDGMENT

The work was partially supported by Grant of President of the Russian Federation MD-1687.2008.9 and Analytical Departmental Program “Developing the scientific potential of high school” 2.1.1/1775.

## REFERENCES

- [1] D. Deutsch. *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proceedings of the Royal Society London, 1985, Vol. A400, 97–117.
- [2] P. W. Shor. *Algorithms for quantum computation: Discrete logarithms and factoring*, Proceedings of the 35th IEEE Symposium on Foundations of Computer Science, 1994. pp.124–134.
- [3] P. Pudlák. *Complexity theory and genetics*, Proceedings of 9th Conference on Structure in Complexity Theory, 1994. pp.183–195.
- [4] L. M. Adleman. *Molecular computation of solutions to combinatorial problems*, Science, 1994, Vol. 266, 1021–1024.
- [5] R. J. Lipton. *Speeding up computations via molecular biology*, Technical report, Princeton University, 1994.
- [6] R. J. Lipton. *Using DNA to solve NP-complete problems*, Technical report, Princeton University, 1995.
- [7] R. J. Lipton. *DNA solution of hard computational problem*, Science, 1995, Vol. 268, 542–545.
- [8] P. W. K. Rothmund. *A DNA and restriction enzyme implementation of Turing machines*, R.J.Lipton and E.B.Baum, editors. DNA Based Computers: Proceedings of the DIMACS Workshop, Princeton University, Providence, Rhode Island, 1996. pp.75–119.
- [9] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, E. Shapiro. *Programmable and autonomous computing machine made of biomolecules*, Nature, 2001, Vol. 414, 430–434.
- [10] Y. Benenson, R. Adar, T. Paz-Elizur, E. Keinan, Z. Livneh, E. Shapiro. *DNA molecule provides a computing machine with both data and fuel*, Proceedings of the National Academy of Sciences of the United States of America, 2003, Vol. 100, 2191–2196.
- [11] Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, E. Shapiro. *An autonomous molecular computer for logical control of gene expression*, Nature, 2004, Vol. 429, 423–442.
- [12] T. H. LaBen, H. Yan, J. Kopatsch, F. Liu, E. Winfree, J. H. Reif, N. C. Seeman. *The Construction of DNA Triple Crossover Molecules*, Journal of the American Chemical Society, 2000, Vol. 122, 1848–1860.
- [13] H. Li, S. H. Park, J. H. Reif, T. H. LaBean, H. Yan. *DNA-Templated Self-Assembly of Protein and Nanoparticle Linear Arrays*, Journal of American Chemistry Society, 2004, Vol. 126, 418–419.
- [14] S. H. Park, H. Yan, J. H. Reif, T. H. LaBean, G. Finkelstein. *Electronic nanostructures templated on self-assembled DNA scaffolds*, Nanotechnology, 2004, Vol. 15, 525–527.
- [15] N. C. Seeman. *Nucleic Acid Junctions and Lattices*, Journal of Theoretical Biology, 1982, Vol. 99, 237–247.
- [16] A. T. Winfree. *The Geometry of Biological Time*, Springer-Verlag, Berlin, 2000.
- [17] H. Yan, T. H. LaBean, L. Feng, J. H. Reif. *Directed Nucleation Assembly of Barcode Patterned DNA Lattices*, Proceedings of the National Academy of Science of the United States of America, 2003, Vol. 100, 8103–8108.
- [18] H. Yan, S. H. Park, G. Finkelstein, J. H. Reif, T. H. LaBean. *DNA-Templated Self-Assembly of Protein Arrays and Highly Conductive Nanowires*, Science, 2003, Vol. 301, 1882–1884.
- [19] P. Yin, A. J. Turberfield, J. H. Reif. *Design of an Autonomous DNA Nanomechanical Device Capable of Universal Computation and Universal Translational Motion*, Tenth International Meeting on DNA Based Computers. LNCS 3384, Springer-Verlag, New York, 2005. pp.426–444.

- [20] M. N. Stojanovic, T. H. E. Mitchel, D. Stefanovic. *Deoxyribozyme-based Logic Gates*, Journal of American Chemistry Society, 2002, Vol. 124, 3555–3561.
- [21] M. N. Stojanovic, P. de Prada, D. W. Landry. *Homogeneous assays based on deoxyribozyme catalysis*, Nucleic Acids Reserch, 2000, Vol. 28, 2915.
- [22] M. N. Stojanovic, D. Stefanovic. *A deoxyribozyme-based Molecular Automaton*, Nature Biotechnology, 2003, Vol. 21, 1069.
- [23] M. N. Stojanovic, D. Stefanovic. *Deoxyribozyme-based Half-Adder*, Journal of American Chemistry Society, 2003, Vol. 125, 6673.
- [24] <http://bdcc.kmpn.net/htmls/dnacomputer/index.php>
- [25] <http://bi.snu.ac.kr/biocomputers2004>
- [26] G. Păun. *Membrane Computing: An Introduction*, Springer-Verlag, Berlin, 2002.
- [27] G. Păun. *Membrane computing: Main ideas, basic results, applications*, Molecular Computational Models: Unconventional Approaches (M. Gheorghe, ed.), Idea Group Publ., London, 2004. pp.1–31.
- [28] G. Felsenfeld, G. Cantoni. *Use of thermal denaturation studies to investigate the base sequence of yeast serine sRNA*, Proceedings of the National Academy of Science of the United States of America, 1964, Vol. 51, 818–826.
- [29] H. Lodish, A. Berk, P. Matsudaira, C. A. Kaiser, M. Krieger, M. P. Scott, S. L. Zipursky, J. Darnell. *Molecular Biology of the Cell*, W.H. Freeman, New York, 2004.
- [30] M. D. Friede. *Design of artificial tRNAs*, Dissertation zur Erlangung des akademischen Grades Doctor rerum naturalium. Vorgelegt der Form- und Naturwissenschaftlichen Fakultät der Universität Wien, Wien, 2001.
- [31] G. Keith and G. Dirheimer. *Evidence for the existence of an expressed minor variant tRNA<sup>phe</sup> in yeast*, Biochemical and Biophysical Research Communications, 142:183–187, 1987.
- [32] J. Weissenbach, I. Kiraly, and G. Dirheimer. *Primary structure of tRNA<sup>thr</sup> 1a and b from brewer's yeast*, Biochimie, 59:381–391, 1977.
- [33] M. Sprinzl, C. Horn, M. Brown, A. Ioudovitch, and S. Steinberg. *Compilation of tRNA sequences and sequences of tRNA genes*, Nucleic Acids Research, 26:148–153, 1998.
- [34] C. R. Woese, O. Kandler, and M. L. Wheelis. *Towards a natural system of organisms: proposal for the domains archaea, bacteria and eucarya*, Proceedings of the National Academy of Sciences of the United States of America, 87:4576–4579, 1990.
- [35] G. R. Bjork, J. M. Durand, T. G. Hagervall, R. Leipuviene, H. K. Lundgren, K. Nilsson, P. Chen, Q. Qian, and J. Urbonavicius. *Transfer RNA modification: in on translational frameshifting and metabolism*, FEBS Letters, 452:47–51, 1999.
- [36] C. G. Edmonds, P. F. Crain, R. Gupta, T. Hashizume, C. H. Hocart, J. A. Kowalak, S. C. Pomerantz, K. O. Stetter, and J. A. McCloskey. *Post-transcriptional modification of tRNA in thermophilic archae (archaeobacteria)*, Journal of Bacteriology, 173:3138–3148, 1991.
- [37] D. A. Melton, E. M. de Robertis, and R. Cortese. *Order and intracellular location of the events involved in the maturation of a spliced tRNA*, Nature, 284:143–148, 1980.
- [38] K. Nishikura and E. M. De Robertis. *RNA processing in microinjected xenopus oocytes. Sequential addition of base modifications in the spliced transfer RNA*, Journal of Molecular Biology, 145:405–420, 1981.
- [39] B. Masquida and E. Westhof. *On the wobble G-U and related pairs*, RNA, 6:9–15, 2000.
- [40] R. W. Holley. *Structure of an alanine transfer ribonucleic acid*, JAMA, 194:868–871, 1965.
- [41] N. B. Leontis and E. Westhof. *Conserved geometrical base-pairing patterns in RNA*, Quarterly Review of Biophysics, 31:399–455, 1998.
- [42] G. A. Burley, J. Gierlich, M. R. Mofid, H. Nir, S. Tal, Y. Eichen, and T. Carell. *Directed DNA Metallization*, Journal of the American Chemical Society, 128(5):1398–1399, 2006.
- [43] A. S. Swami, N. Brun, and D. Langevin. *Phase Transfer of Gold Metallized DNA*, Journal of Cluster Science, 20(2):281–290, 2009.
- [44] E. A. Burakova and V. N. Silnikov. *Molecular Design of Artificial Ribonucleases Using Electrostatic Interaction*, Nucleosides, Nucleotides and Nucleic Acids, 23(6-7):915–920, 2004.
- [45] D. Rooß and K. W. Wagner. *On the Power of DNA-Computing*, Information and Computation, 131(2)95–109, 1996.



# Using Hardware-Assisted Virtualization to Protect Application Address Space Inside Untrusted Environment

Denis Silakov

Institute for System Programming  
at the Russian Academy of Sciences  
Moscow, Russian Federation  
Email: silakov@ispras.ru

**Abstract**—In this paper we present a virtualization-based approach of protecting execution of trusted applications inside potentially compromised operating system. In our approach, we do not isolate application from other processes in any way; instead, we use hypervisor to control processes inside OS and to prevent undesired actions with application resources. The only requirement for our technique to work is presence of hardware support for virtualization; no modifications in application or OS are required.

**Index Terms**—Virtual Machine Monitor, Hypervisor, Security, Protection

## I. INTRODUCTION

In modern software world, an operating system is a key component responsible for many security aspects of application execution process. In particular, it should provide possibilities to manage access permissions of application files and other resources, guarantee isolation of application address space in memory and so on.

However, many widespread operating systems (such as Linux or Windows) are known to be subjected to vulnerabilities which can be used by malicious code to compromise the whole system or particular application. As operating systems evolve, vulnerabilities are detected and fixed. But at the same time a lot of new features are added which potentially introduce new vulnerabilities. Size of code which is executed with highest privileges in modern OS is large. In particular, many popular systems are based on monolithic kernel where every device driver is a part of the kernel (that is, works in the same address space with other kernel parts and other drivers). It is common for drivers to contain issues. Vulnerability research performed in 2005 has shown that device drivers were responsible for about 85% of failures in Windows XP [1]; similar statistics was reported for Linux [2]. It is very likely that the situation will not change in the near future, since size of drivers grows faster than size of any other part of the kernel [3].

Microkernel-based operating systems are claimed to be more secure due to the fact that the size of code executed in privileged mode is very small [4]. However, in such systems interaction between micro kernel and drivers which

becomes quite expensive. If used on a desktop machine with lots of peripheral devices, such systems often demonstrate worse performance. In addition, there are a lot of applications developed for widespread OSes with monolithic kernels. It would be very expensive to port all these programs to a system with completely new architecture. As a result, nowadays microkernel-based systems are primarily used either in highly tailored areas (e.g., QNX for embedded real-time systems) or for educational purposes (e.g., Minix).

Thus, there is a need for application protection techniques that will not require modifications of existing operating systems or applications, but at the same time will provide more reliable and secure services than traditional approaches.

One of the possible techniques is to use hardware-assisted virtualization. As implemented in modern Intel and AMD processors, it allows to launch a program (called *hypervisor*) that has full control over hardware and runs with higher privileges than OS. Normally, hypervisor is responsible for virtualization (e.g., creating and managing virtual machines), but its functionality can be enhanced. In particular, it can provide some security services. Hypervisor is usually much smaller than OS (for example, most hypervisors do not have a large set of device drivers). As a result, hypervisor potentially contains less vulnerabilities and usually considered to be more secure than commodity operating systems. In this paper, we suggest an approach for protecting application address space using hypervisor.

The remainder of the paper is structured as follows: Section 2 observes existing virtualization-based approaches to protection of application resources. Section 3 describes general architecture of our protection system and specific aspects of protecting address space of applications of different kinds. Section 4 describes implementation of our approach and present performance measurement results. Finally, Section 5 summarizes the main ideas.

## II. HYPERVISOR-BASED PROTECTION SYSTEMS

The idea of using hypervisor for different security tasks has got much attention after Intel and AMD introduced their first implementations of hardware-assisted virtualization in

years 2005-2006. Many approaches requires modifications of applications, OS (e.g., [5] or [6]) or even hardware ([7], [8]). Though some of these approaches seem to be quite effective, their usage is rather limited.

A promising approach is Overshadow technology of memory protection suggested by researches from Stanford and Princeton Universities, MIT and VMware, Inc ([9]). It does not require modifications of OS or applications. Instead, it encrypts process memory area of working processes. If OS or other program try to access process memory, they only see encrypted data. For trusted process itself, a "normal" memory view is provided. Similar approach based on dynamic encryption of application memory is presented in [10].

However, these approaches are primarily aimed at hiding application data from third parties. In our work, we suggest an approach that allows other processes to read memory of a trusted process, but denies to modify it. Such assumption is useful for cases when trusted application needs to pass some data to other processes by means, for example, of shared memory. That is, our approach protects execution process of a trusted application, but does not hide its whole data from other programs. But if necessary, our system can be easily modified to completely deny access to application's memory.

An advantage of Overshadow is that no modifications are required in existing software (OS, applications) and hardware. More precisely, there are no hardware-specific requirements only if hypervisor used is able to perform virtualization without hardware assistance. However, in this case protection system architecture is bounded to architecture of particular hypervisor. Moreover, such hypervisors for x86 platform are rather complex and they are rather hard to implement (since x86 architecture by itself is hard to virtualize due to design). Among effective implementations, we can mention only VMware VMM (used in Overshadow) and VirtualBox [11]. Since such hypervisors are complex (and in addition, VMware hypervisors are mostly closed source products), it is not easy to modify them to implement additional functionality.

On the other hand, in the last several years Intel and AMD have added virtualization support to their processors and made it easier to create virtualization products [12]. These possibilities are now utilized by such products as Kernel-based Virtual Machine (KVM), Xen, VMware ESX and others. In our approach we assume that target system provide hardware-assisted virtualization. This puts some limitations on hardware where our approach is applicable, but significantly simplifies its implementation.

### III. CONTROLLING CONSISTENCY OF A TRUSTED PROCESS

In our threat model, we suppose that the operating system is not reliable and contains vulnerabilities which can be exploited by malware to gain high privileges. Such privileges would allow attacker to control all processes running in the system and perform malware injections in their files or directly in the process code in the memory.

In our protection system, potentially compromised OS is located inside virtual machine controlled by hypervisor, which is a core part of the protection system. Hypervisor has higher privileges than OS inside VM and can monitor and control events inside VM.

In order to guarantee consistency of a trusted application, hypervisor should guarantee the following:

- application files on disk (in particular, executables and libraries) are not modified by malicious software; in this paper, we only consider executable files and libraries that form the application, ignoring the task of protection of other files and resources that can be used by application (e.g., protection of different data files);
- address space of a running process is not modified in an unallowed way by other processes running in OS.

Let us consider how these tasks are solved by suggested protection system.

#### A. Checking Consistency of Executable Files and Libraries

When launching a trusted application, we should first ensure that executable being launched is an expected one. In order to do this, we should check that application executable file (and shared libraries, if any) on disk was not modified by malicious code. To make such check possible, every trusted application in our system should provide hypervisor with a *registration data*, generated inside trusted environment on the basis of application files. This registration data is stored in hypervisor and cannot be accessed by OS.

Registration data for application executable files and shared libraries consists of SHA-1 hash codes. Such codes are generated for every memory page containing either instructions or static data. Currently we assume that the page size is equal to 4 kilobytes (a default value on most systems). However, nowadays Linux provides support for larger pages [13], and we plan to support such pages in future, as well.

#### B. Protecting Control Flow

In our system, the working virtual machine is provided with a single-core virtual CPU, so OS inside this VM can only use a pure time-sharing multitasking. There is no way to run different processes on different CPU cores in parallel. In particular, at any moment of time either CPU and other resources are used by trusted code or they are used by potentially malicious software. Thus, if we want to protect trusted process, we should only ensure that the process address space and other system resources that can influence process execution (e.g., different system registers) were not modified in a forbidden way while the trusted process was inactive. When trusted process is active, all events in the system are allowed. In particular, trusted process can modify its own code segments in memory. Besides application code and static data loaded at launch, we can control consistency of any data pages loaded by during process execution. More particular, we track states of all memory pages written by the process.

In order to implement such protection, we use hypervisor to handle interruptions of trusted code execution. When a trusted

process is interrupted, hypervisor saves information about its address space and other protected resources inside its own memory. Only after that, control is passed to operating system. When OS returns control to the trusted process, hypervisor compares actual state of protected resources with the saved one. If any discrepancy is detected, the protection system reports an attack attempt and the process is not considered to be trusted any more. From that moment, it will not be allowed to use protected system resources (e.g., network connection).

One of the main components of the protection system is a *register integrity checker* used to protect control flow of trusted processes. The control flow is considered to be integral, if the following requirements are met:

- 1) actual address of program entry point is equal to the value specified in the registration data;
- 2) every time the control is passed from the OS kernel to the trusted process, address of instruction invoked in the process is either equal to the instruction where the process was previously interrupted, or is equal to a special signal handler (registered by the process in advance).

The first requirement is checked only when the process is launched using a system call like *exec()*. More generally, it should be checked when the process enters the trusted mode, but in our work we do not consider situations when the process can enter the trusted mode after the launch. The second requirement is checked every time the control is passed to the trusted process. When such an event occurs, the hypervisor verifies instruction address, as well as values of general purpose, segment and different system registers.

### C. Protecting Address Space

Hypervisor controls integrity of all virtual memory pages (containing either code or data) of the process. When a trusted process accesses a memory page for the first time, this page is marked as *active*. If the page accessed for the first time contains program code or static data, then it is checked that the page hash sum corresponds to the one specified in the registration data. This allows to verify that the program code and static data were not modified after registration data was generated. Other pages are allowed to have random data when they are accessed by trusted process for the first time. If in the sequel trusted process accesses such a page, the hypervisor checks that the page content was not modified since the last time when it was accessed by the process.

In order to perform such integrity monitoring, hypervisor uses a special control table of process active virtual pages which we call *Memory Integrity Table* (MIT). For every virtual page  $V$ , the MIT table contains either address of corresponding physical page  $P$  (if  $V$  is mapped to a physical memory) or hash sum  $H$  if the page is not mapped.

At runtime, programs can detach memory pages from their address space (e.g., by means of *munmap()* system call). Hypervisor tracks such system calls and removes the *active* mark from the detached pages.

Pages storing dynamic data inside address space of a trusted process can be subjected to legal modifications by the process itself, as well as by some system calls (e.g., *read()*). If an active page of a trusted process is mapped to a physical page, then write access to that page is allowed for the trusted process only. When a trusted process tries to access a page for which a hash sum is set in the MIT table, hypervisor checks integrity of that page by calculating hash code for its current content and comparing it with the expected value stored in the control table.

Moreover, hypervisor allows only modifications that touch memory areas explicitly specified in the system call parameters. Modifications outside such explicitly specified areas are prohibited. It is important to note that on Intel x86 architecture it is possible to set access permissions on the page-level basis, while processes may want to write data which is not aligned to page size. In order to support protection of such data, hypervisor used special trick based on the fact that for every process one can specify address area writable for kernel with per-byte precision.

Before transferring system call to OS kernel, the protection system for every out parameter allocates a "shadow" memory area inside virtual address space of the process and set registers controlling passing of return values to point to that area. Thus, output of every system call is redirected to memory area not used by the process. When system call returns control to the process, hypervisor copies its output to corresponding areas inside process memory.

In order to maintain mappings in the MIT table and to intercept page access attempts, hypervisor runs every trusted process in a separate *protection domain*. Protection domain is a set of memory pages with individual access permissions. This set of pages for a particular protection domain is dynamically altered by hypervisor when process requests more memory or frees unnecessary pages. Every attempt to access a page outside the protection domain, as well as access violation for the page inside the domain, leads to exception which is caught and handled by hypervisor.

Implementation of protection domains is based on the *Nested Page Tables* (NPT) mechanism (NPT implementation in Intel processors is called *Extended Page Tables*, the one from AMD – *Rapid Virtualization Indexing*). NPT tables are used to perform translation of pseudo-physical addresses used inside VM to physical addresses of the real hardware. When a process is launched in the trusted mode, hypervisor creates an empty set of NPT tables for it. Every time when OS kernel passes control to the trusted process, hypervisor activates page tables corresponding to that process. This is performed by means of the *Virtual Machine Control Block* (VMCB) structure. When trusted process is interrupted and control is passed back to the OS kernel, hypervisor switches active nested pages once again and activates tables of untrusted domain (a joint domain for OS kernel and other untrusted processes).

When a process tries to access a page which is not yet reflected in the NPT tables, or when access violation occurs, a

Nested Page Fault (#NPF) exception is thrown, VM is stopped and control is passed to the hypervisor. Hypervisor maintains NPT mapping only for active pages which are not swapped out to the storage device and which were not modified by third-party processes. This approach allows to determine if the process accesses a page for the first time or it accesses pages which were modified since the last access by this process or loaded from swap.

When the #NPF exception is thrown, a pseudo-physical address of page inside VM is reported, access to which led to the exception. However, in order to get the expected hash sum for the page from the control tables, hypervisor should also know a virtual address, access to which finally led to #NPF. In order to calculate virtual address, hypervisor disassembles the current instruction of the trusted process (address of such instruction is always stored in the IP register) and analyzes all virtual addresses accessed by this instruction. Using page table of the operating system, hypervisor calculates real addresses corresponding to these virtual ones and detects which of them corresponds to the pseudo-physical address access to which led to the #NPF exception. With this virtual address, the hypervisor is able to verify integrity of the page accessed by the trusted process.

#### D. Protecting Dynamically Linked Applications

Address space protection approach described above easily applies for statically linked programs. Such a program is represented by a single executable file that does not import any libraries from the OS, so we can know in advance location of code and static data inside the application. However, nowadays developers often take an advantage of using splitting functionality between separate libraries which are combined together by the loader during program start up or even loaded by request during program execution (such functionality is provided in Linux by `libdl` library). Protection of such programs (especially those that use `libdl` functionality) introduces new challenges.

Dynamically linked application consists of a main executable file and several libraries loaded by dynamic loader during application launch. In Linux, for both executable files and dynamic libraries ELF format is used. Every ELF file has a set of `DT_NEEDED` entries which store names of libraries that should be loaded with this file. When launching an executable, dynamic loader processes `DT_NEEDED` entries of the file itself, then `DT_NEEDED` entries of libraries loaded as file dependencies and so on – such iterations are performed until `DT_NEEDED` entries of all files from the loaded set are satisfied by files from this set.

The set of `DT_NEEDED` entries can be extracted from the ELF file by means of appropriate tools. However, the final set of loaded libraries can be different for the same executable in different Linux distributions, because internal dependencies of libraries can differ. Moreover, in addition to dependencies statically recorded in the ELF file structures which are resolved during file launch, it is possible to load libraries at runtime by means of functions provided by `libdl` library. In many cases,

it is almost impossible to automatically detect which libraries will be loaded using such functionality, because the name of the library to be loaded can be calculated at runtime.

Due to these facts, in our approach user should explicitly list all the libraries that will be loaded during application work in particular system. This set considered to be a set of trusted files. If a library not included in this set will be loaded and put to the application address space, this will be reported as an attack attempt.

In addition to libraries, for every dynamically linked application the Linux kernel exposes a shared object called *Virtual Dynamically-linked Shared Object* (VDSO) which exports symbols implementing virtual system calls [14]. Traditionally, system calls in Linux on the x86 platform were implemented using 0x80 software interrupt. With modern processors, faster implementations are available that use `SYSCALL` or `SYSENTER` instructions for AMD and Intel processors respectively. For every of these techniques, the Linux kernel has a corresponding VDSO variant. Implementation of all these three VDSO variants can be extracted from the Linux kernel sources.

Thus, a memory image of a dynamically linked application consists of the following components:

- executable file (launched by user or by other process);
- dynamic loader (usually – `ld-linux.so`);
- set of libraries specified as ELF file dependencies and loaded at application start;
- set of libraries loaded at runtime using `libdl` functionality;
- VDSO library.

Registration data of dynamically linked application should contain information about all these components.

An important feature of dynamic libraries is that their code is position-independent and can be located any area of application's address space. Address value specified in the ELF file header in Linux running on x86 platform nowadays is just a recommendation for the loader. In reality, dynamic loader can place every file at other address, and such addresses can vary in different systems or even in different instances of the same application.

Note that since VDSO is a shared object, it can also be located at any address inside process address space. Thus, location of VDSO in process memory can be different for different processes.

Finally, executable files can also contain position-independent code. Executable files that consist of such code only (*Position-Independent Executables*, PIE) are relatively widespread in the Linux ecosystem.

Thus, every component of dynamically linked application can be located at any virtual address inside application address space. Since the memory is allocated and managed on the per-page basis, correlation between actual address of every component and the value specified in ELF header is expressed by the following formula:

$$Actual\_address = ELF\_address + k * (page\_size)$$

where  $k$  is some integer number.

Thus, though location of different components of dynamically linked application in virtual memory is not known in advance, these locations can be easily calculated by hypervisor during application start up. Location of libraries loaded using `libdl` functions can be calculated at the moment when `dlopen()` function is invoked. This allows to adopt registration data for every particular launch of application. As we will discuss later, the only thing hypervisor has to calculate is a difference between real address and the value specified in the ELF header (and thus reflected in the registration data) which is identified by a single integer number  $k$ . It is important to note that since library code is position independent, it is not subjected to any modifications by loader.

Before passing control to entry point of dynamically linked application, dynamic loader should link together all components of application and set actual addresses of all imported symbols. For the context protection system, it is important to ensure that no malicious code interfere with this process, replacing address of legal imported function with address of malicious symbol. Let us proceed with details of dynamic loader work process and see how protection system guarantees consistency of function addresses.

During dynamic linking process, system loader first loads all necessary files to memory and then initiates symbol resolution process. For every binary symbol imported by some ELF file (this file is called *importer*) the loader should locate the file where the symbol is implemented (this file is called *exporter*). Dynamic loader analyzes symbol tables of exporter and importer and updates the Global Offset Table (GOT), which is located at the data segment of the importer. The GOT table contains an entry for every imported binary symbol (corresponding to a function or global variable). Symbol resolution procedure is the same for libraries loaded during application launch and the one loaded at runtime using `libdl` functionality.

Code segment of ELF file that imports some functions contains Procedure Linkage Table (PLT) which contains a *stub* symbol for every imported function. When an attempt is performed to call some imported function, the control is passed to the corresponding stub which takes unnecessary address from the GOT table and passes control to that address. Thus, a call to an imported function is an indirect call by address recorded in the appropriate GOT table entry.

GOT and PLT tables used when a call to imported function happens are located in the segments of that ELF file from which the call is performed. Thus, they are taken into account when file hash sum is calculated and monitored by the protection system during application work. The whole dynamic linking process, including modification of the GOT table, is performed by the dynamic loader which works in the user space. The process requires no kernel-level activities and thus cannot influence other applications in case of errors. This is one of the advantages of using ELF format for executable files.

Thus, if dynamic loader is a trusted program, then all the actions during dynamic linking are performed by trusted code. Dynamic loader is much more smaller than the Linux kernel and it does not vary significantly among different distributions

(in particular, it does not allow insertion of some third-party software such as drivers in its code). Thus, we believe that it is reasonable to consider dynamic loader to be a trusted process. In the rest of the paper, we use the assumption that the dynamic loader is a trusted program. Note that since dynamic loader is included in the process image, hypervisor is able to compare its content with registration data. Thus, the protection system is able to check that the loader is the same as the one in the system where the registration data was generated.

When launching a dynamically linked application, the Linux kernel creates a virtual address space for the new process and loads application executable file (which was actually launched), dynamic loader and VDSO library there. All other libraries are loaded using explicit calls to the `mmap()` system call from the dynamic loader. This call returns a virtual address where the library is located. Since hypervisor monitors all system calls performed by application, it can track library loading and build a mapping between library name (passed as a parameter to the `open()` call, whose result is then passed to `mmap()`) and library location in the process address space. For every loaded library it is checked, if the library is included in the list which was provided by user when generating registration data. If so, then hypervisor is able to compare hash sum of the loaded library with the expected value and verify that this is, indeed, an expected file. If the loaded file is not included in the list of trusted libraries, or if its hash sum does not match the value expected, then the loaded code reported to be untrusted. If the control is passed to such code, then hypervisor will nullify application privileges, so it will not be considered to be trusted any more.

OS kernel passes addresses of components loaded during application launch to dynamic loader using *ELF auxiliary vectors* [15]. During application launch, array of such vectors is put at the process stack just after environment variables and thus can be easily analyzed by hypervisor. Each vector is just a pair of numbers (vector *type* and *value*).

In order to be able to protect application address space, we should know values of vectors with the following types:

- `AT_PHDR` – Base address of executable file;
- `AT_ENTRY` – Entry point of program ;
- `AT_BASE` – Base address of dynamic loader;
- `AT_SYSINFO_EHDR` – Base address of the VDSO library.

When application is launching, the control is first passed to the entry point of the dynamic loader. The loader performs dynamic linking of the executable file launched and libraries loaded as its dependencies and then passes control to the address specified in the `AT_ENTRY` vector. `AT_SYSINFO_PHDR` vector is used by `libc` library to perform a system call.

Every component of dynamically linked application can be loaded at (almost) random address which differs from the one recorded in the file header. However, in any case the following conditions are met:

- For every loaded component, hypervisor can obtain the effective address where the component is located in

virtual memory before the control is passed to that component or before that component is accessed by someone else (e.g., by dynamic loader which should at least read header of loaded file during dynamic linking process). Thus, hypervisor can verify component integrity before the component is used by other parts of trusted application.

- Every file is stored in the address space continuously, so if a file is loaded at the address different from the one specified in its header, hypervisor just have to shift registration data for this file, without a need to recalculate it.

Thus, hypervisor is still able to protect application address space, but for every application component it should calculate an effective address where the component is loaded. For executable files, dynamic interpreter and VDSO library such addresses can be obtained from `AT_PHDR`, `AT_BASE` and `AT_SYSINFO_EHDR` vectors respectively. For other libraries, the effective address is a result of the `mmap()` system call.

After calculation of real address value, hypervisor adjusts registration data for the trusted process by updating the MIT table which stores mapping between addresses of every page of trusted process and hash codes. For every component, such an update is performed as soon as component virtual address becomes known. For shared libraries, this happens after return from the `mmap()` system call (for libraries specified directly as ELF file dependencies – during application start up, for libraries loaded using `dlopen()` function – when loading a library during application work). For other components the update happens at the moment of application start up (more particular – after return from the `execv()` system call).

Information about real location of application components is obtained from the Linux kernel, which is untrusted in our threat model. However, if kernel provides hypervisor with wrong information (that is, real address values differ from the one reported by kernel), this will be detected as soon as some part of application will try to access a page with wrong data. In this case, expected hash sum for the page will differ from the observed one and the attack will be reported. Thus, it is impossible for the kernel to substitute some part of the application without being noticed.

It is important to note that similar to hypervisor, dynamic loader during the linking process uses information obtained from the kernel. Protection system should guarantee that the loader uses the same data as the hypervisor itself – otherwise the resulting application image in memory can differ from hypervisor expectations. Thus, we should guarantee that hypervisor and dynamic loader use the same values of auxiliary vectors. But these vectors are located in the application stack which is monitored by our protection system and whose integrity is guaranteed. Thus, hypervisor is able to control that the loader itself (`ld-linux.so`) and application components (executable and libraries) are consistent and match registration data.

Thus, protection of stack, code segment and process data automatically guarantees protection of PLT and GOT tables,

as well as ELF auxiliary vectors.

In order to prevent execution of unauthorized code, hypervisor uses one more feature of modern hardware architectures, namely NX (No eXecute) bit. If a process tries to execute instruction from a page marked with this bit, a page fault exception is thrown. Hypervisor sets NX bit in the NPT for all pages of a trusted process except those that contain executable code authorized by means of registration data. If a trusted process attempts to execute a code from a NX page, a page fault exception is thrown. In case of page fault, hypervisor checks error code and if the fault was caused by attempt of launching some instruction, then the process is considered to be compromised and protection system deprives this process of privileged rights, so it is not trusted any more.

### E. Protecting MultiThread Applications

In addition to dynamically linked programs, nowadays many applications use multithread paradigm, when program consists of several threads which work in parallel in the same address space. In Linux, threads are created using `clone()` system call with `CLONE_VM` flag.

All threads of the same process have the same page tables. For such threads, hypervisor also uses the same nested page tables and the control MIT table. It is important that in our system a virtual machine where the trusted processes work has only one CPU core available, so at any time point only one thread can really execute CPU instructions. Thus, when considering address space access, we can safely ignore the fact that the process is divided on several threads. In a single-core system, it does not matter if the access is performed from mono-thread trusted process or from some thread of multithread trusted process. In addition, absence of “real” parallelism allows us not to care about synchronization of thread access to nested page tables or MITs.

However, every process thread in Linux can have its own address space to store data unique to particular thread. This storage (called *thread local storage*) is created automatically by compiler for variables that have `__thread` specifier, or can be created at runtime by means of functions like `set_thread_area()`.

Thread local storage is implemented completely on the software layer by compiler, libc library and Linux kernel. For every thread, a separate memory area is allocated to store thread-specific data. Since all this memory areas are allocated inside process address space (common for all threads), every thread can potentially access a local storage of any other thread.

Thread local storage is located in a separate segment managed by the GS register which is set to different values for different threads when thread-local data is accessed. In order to control integrity of thread local data, the memory protection system should monitor GS register value and corresponding record in the segment table. These values can be monitored in the same way as other resources and processor registers, so protection of multithread applications fits well the approach used in our protection system.

## IV. IMPLEMENTATION

The approach suggested in this paper was implemented on the basis of KVM (Kernel-based Virtual Machine) hypervisor which is included in the Linux kernel. KVM itself is a kernel module which adds hypervisor functionality to the Linux kernel. KVM requires QEMU application to manage and emulate different virtual machine resources and devices (keyboard, network card, etc.). In our work, we use KVM version 88, kernel 2.6.31.6 and QEMU 0.13.0. Virtual machines with trusted applications are run under Fedora Linux 13 with the same kernel (2.6.31.6). Currently our implementation supports virtualization of 32-bit systems on AMD platform. For the experiments described below, we have used AMD Phenom 9750 Quad-Core Processor which has four 2.4MHz cores. The host machine had 4GB RAM, and virtual machine was configured with 512MB RAM. It is important to note that it is not necessary to load hypervisor when the machine starts; we have investigated possibility of on-demand activation of protection system [16]. Such on-demand activation (which involves launching hypervisor from running OS, creation of a virtual machine and placing the OS inside this machine) is possible, though requires special hardware (Trusted Platform Module, TPM).

To automate generation of registration data for trusted processes, we provide a tool named ElfHash that processes executable file which will be launched and its DT\_NEEDED dependencies. We suppose that the system where the registration data is generated provides trusted versions of libraries used by dynamically linked application. On the basis of such assumption, the ElfHash tool analyzes system libraries that satisfy application dependencies and create registration data for them. Alternatively, user can provide the tool with his own versions of such libraries. In additional, user can specify libraries not mentioned in file dependencies but that can be loaded at runtime using `libdl` functionality. Finally, we provide registration data for the dynamic loader itself. The data is generated using VDSO implementation which is considered to be trusted.

### A. Attack Detection

In order to evaluate if the protection system works as expected, we have emulated two kinds of attacks on trusted processes: modification of application files on disk and modification of trusted code in memory. For our experiments, we have used the SSH tool which was establishing connection from the virtual machine to some remote host. Network card was considered to be a protected system resource, so only trusted applications were allowed to access it. Operating system itself was not aware of network card.

When emulating attacks concerning application file modifications, we have investigated behavior of the protection system in the following cases:

- SSH executable differs from the one registered in the protection system;

- one of the libraries from ELF DT\_NEEDED dependencies differs from the one used to register SSH in the protection system;
- one of the libraries loaded by SSH using `dlopen()` differs from the one registered in the protection system.

In the first two cases, the protection system reported the attack attempt during application startup. In the third situation, attack attempt was reported at the moment of `dlopen()` call. In all cases, access to network card was denied and connection to remote host was not established.

In addition, we have checked situations when library is loaded which is not present in registration data. The library can either be loaded by the process itself or pre-loaded if user sets `LD_PRELOAD` variable. In such situations, the protection system also reported attack attempt, as expected.

To emulate attacks concerning modification of the code of a running process, we have used techniques based on the `ptrace()` system call. In particular, we have used the PreZ tool [17] which attaches to running process and creates its own thread inside it. This thread opens a port for TCP connections and spawns a shell for every incoming connection. The shell can then be used by remote party to perform different actions on the machine with privileges of the infected process.

During our experiments, the protection system has successfully detected all code injection attempts and blocked access to the network card for the SSH process.

### B. Performance

In order to estimate delays introduced by the protection system, we have compared performance of two applications – Apache web server and SSH client – in the following cases:

- applications are launched on bare hardware;
- applications are launched inside virtual machine without protection systems;
- applications are launched inside virtual machine with protection system controlling their address space.

In order to measure Apache performance, we have used the Flood load tester (a part of the Apache project). Number of processes launched by Apache to serve the requests (that is, number of trusted processes) was limited to ten. In case of SSH, we have used the SCP utility to copy large (four gigabytes) file through network. There was only one trusted process in this experiment.

Since our protection system assumes that only one processor core is assigned to virtual machine, in our experiments we were using a single core in all cases. Measurement results are presented at Fig.1. We have normalized the results and assigned 100 units to performance on the bare hardware, so it is easy to compare the measurements.

As one can see, in case of Apache performance loss is almost unnoticeable, while for SSH it is much more higher. This is probably caused by the fact that during the experiment Apache was receiving simple requests and their processing did not require much memory, while copying file with SSH involved encryption of large amount of data which led to significant usage of memory by trusted process. In addition,

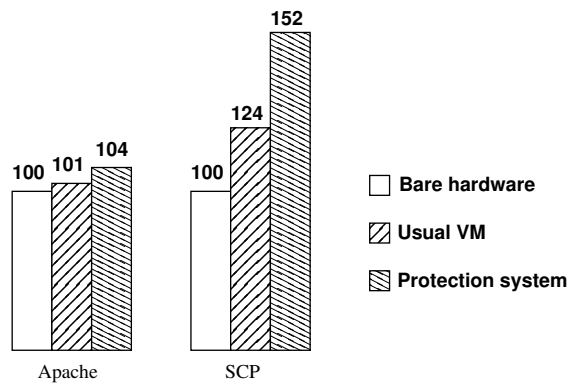


Fig. 1. Protection system performance

when dealing with large data transferring through network, a significant delay is introduced by emulation software by itself. Nevertheless, even for SSH performance loss is acceptable if sender wants to protect the data from potentially compromised OS.

## V. CONCLUSION

In this paper, we have presented a novel approach for protecting applications running inside potentially compromised operating system. The approach is based on using virtual machine monitor (hypervisor) which tracks all events inside OS and prevents unallowed modifications of application resources.

Unlike other hypervisor-based protection techniques, our approach does not lead to isolation of application from other OS components. Hypervisor's functionality is flexible and can be adjusted to control usage of any particular hardware resources, so only trusted applications have access to them. For example, application of our protection system to control usage of network connection is described in [18]. Finally, the approach can be extended in order to protect all application files, not only executables and libraries. This will require interception of direct file input/output (using `read()/write()` system calls) in addition to `mmap()` manipulations and storing hash codes for all files used by application. We believe that it is not hard to extend our approach in this way, though such improvements can introduce significant performance drop.

The approach does not require any modifications in operating system or applications, but relies on several aspects of hardware-assisted virtualization implemented in Intel and AMD x86 processors. In order to implement the approach, there is no need to develop a hypervisor from scratch. Instead, one can extend existing products such as KVM or Xen. Our KVM-based implementation has demonstrated that performance overhead introduced by the protection system is acceptable, so we believe that the approach is viable and can be applied in those areas where information security is the primary goal.

## REFERENCES

[1] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," *ACM Trans. Comput. Syst.*, vol. 23, no. 1, pp. 77–110, 2005.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *SOSP*, 2001, pp. 73–88.

[3] G. Kroah-Hartman, "How linux supports more devices than any other os, ever," *O'Reilly Media Interview*, Oct. 2008. [Online]. Available: <http://broadcast.oreilly.com/2008/10/how-linux-supports-more-device.html>

[4] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, vol. 39, pp. 44–51, May 2006. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1137232.1137291>

[5] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: a virtual machine-based platform for trusted computing," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 193–206, October 2003. [Online]. Available: <http://doi.acm.org/10.1145/1165389.945464>

[6] R. Ta-Min, L. Litty, and D. Lie, "Splitting interfaces: making trust between applications and operating systems configurable," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 279–292. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1298455.1298482>

[7] J. S. Dvoskin and R. B. Lee, "Hardware-rooted trust for secure key management and transient trust," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 389–400. [Online]. Available: <http://doi.acm.org/10.1145/1315245.1315294>

[8] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dvoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," *SIGARCH Comput. Archit. News*, vol. 33, pp. 2–13, May 2005. [Online]. Available: <http://doi.acm.org/10.1145/1080695.1069971>

[9] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dvoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 2–13, March 2008. [Online]. Available: <http://doi.acm.org/10.1145/1353535.1346284>

[10] J. Yang and K. G. Shin, "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 71–80. [Online]. Available: <http://doi.acm.org/10.1145/1346256.1346267>

[11] Oracle vm virtualbox. [Online]. Available: <http://www.oracle.com/us/technologies/virtualization/oraclevm/061976.html>

[12] J. Fisher-Ogden. (2006) Hardware support for efficient virtualization. [Online]. Available: <http://www.cse.ucsd.edu/~jfisherogden/hardwareVirt.pdf>

[13] R. Krishnakumar, "Hugetlb - large page support in the linux kernel." *Linux Gazette*, vol. 155, Feb. 2008. [Online]. Available: <http://linuxgazette.net/155/krishnakumar.html>

[14] J. Petersson, "What is linux-gate.so.1?" Aug. 2005. [Online]. Available: <http://www.trilithium.com/johan/2005/08/linux-gate/>

[15] M. Garg, "About elf auxiliary vectors." 2006. [Online]. Available: <http://articles.manugarg.com/aboutelfauxiliaryvectors.html>

[16] D. Yefremov and P. Iakovenko, "An approach to on-demand activation and deactivation of virtualization-based security systems," in *Proceedings of the fourth Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2010)*, 2010, pp. 157–161. [Online]. Available: [http://syrcose.ispras.ru/2010/files/syrcose10\\_submission\\_5.pdf](http://syrcose.ispras.ru/2010/files/syrcose10_submission_5.pdf)

[17] F. Loukos. (2010) Injecting code at a running process. [Online]. Available: <http://fotis.loukos.me/blog/?p=145>

[18] I. Burdonov, A. Kosachev, and P. Iakovenko, "Virtualization-based separation of privilege: working with sensitive data in untrusted environment," in *Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, ser. VDTs '09. New York, NY, USA: ACM, 2009, pp. 1–6. [Online]. Available: <http://doi.acm.org/10.1145/1518684.1518685>



# Background Optimization in Full System Binary Translation

Roman A. Sokolov  
MCST CJSC  
Moscow, Russia  
Email: roman.a.sokolov@gmail.com

Alexander V. Ermolovich  
Intel CJSC  
Moscow, Russia  
Email: karbo@pvk13.org

**Abstract**—Binary translation and dynamic optimization are widely used to provide compatibility between legacy and promising upcoming architectures on the level of executable binary codes. Dynamic optimization is one of the key contributors to dynamic binary translation system performance. At the same time it can be a major source of overhead, both in terms of CPU cycles and whole system latency, as long as optimization time is included in the execution time of the application under translation. One of the solutions that allow to eliminate dynamic optimization overhead is to perform optimization simultaneously with the execution, in a separate thread. In the paper we present implementation of this technique in full system dynamic binary translator. For this purpose, an infrastructure for multithreaded execution was implemented in binary translation system. This allowed running dynamic optimization in a separate thread independently of and concurrently with the main thread of execution of binary codes under translation. Depending on the computational resources available, this is achieved whether by interleaving the two threads on a single processor core or by moving optimization thread to an underutilized processor core. In the first case the latency introduced to the system by a computational intensive dynamic optimization is reduced. In the second case overlapping of execution and optimization threads also results in elimination of optimization time from the total execution time of original binary codes.

## I. INTRODUCTION

Technologies of binary translation and dynamic optimization are widely used in modern software and hardware computing systems [1]. In particular, dynamic binary translation systems (DBTS) comprising the two serve as a solution to provide compatibility between widely used legacy and promising upcoming architectures on the level of executable binary codes. In the context of binary translation these architectures are usually referred to as source and target, correspondingly.

DBTSs execute binary codes of source architecture on top of instruction set (ISA) incompatible target architecture hardware. They perform translation of executable codes incrementally (as opposed to whole application static compilation) interleaving it with execution of generated translated codes. One of the key requirements that every DBTS has to meet is that the performance of execution of source codes through binary translation is to be comparable or even outperform the performance of native execution (when executing them on top of source architecture hardware).

Optimizing translator is usually employed to achieve higher DBTS performance. It allows to generate highly efficient target

architecture codes fully utilizing all architectural features introduced to support binary translation. Besides, dynamic optimization can benefit from utilization of actual information about executables behavior which static compilers usually don't possess.

At the same time dynamic optimization can imply significant overhead as long as optimization time is included in the execution time of application under translation. Total optimization time can be significant but will not necessarily be compensated by the translated codes speed-up if application run time is too short.

Also, the operation of optimizing translator can worsen the latency (i.e., increase pause time) of interactive application or operating system under translation. By latency is meant the time of response of computer system to external events such as asynchronous hardware interrupts from attached I/O devices and interfaces. This characteristic of a computer system is as important for the end user, operation of hardware attached or other computers across network as its overall performance. *Full system* dynamic binary translators have to provide low latency of operation as well. Binary translation systems of this class target to implement all the semantics and behavior model of source architecture and execute the entire hierarchy of system-level and application-level software including BIOS and operating systems. They exclusively control all the computer system hardware and operation. Throughout this paper we will also refer this type of binary translation systems as virtual machine level (or VM-level) binary translators (as opposed to application-level binary translators).

One recognized technique to reduce dynamic optimization overhead is to perform optimization simultaneously (concurrently) with the execution of original binary codes by utilizing unemployed computational resources or free cycles. It was utilized in a number of dynamic binary translation and optimization systems [2], [3], [4], [5], [6], [7], [8]. We will refer this method as *background optimization* (as opposed to *consequent optimization*, when optimizing translation interrupts execution and utilizes processor time exclusively unless it completes).

The paper describes implementation of background optimization in a VM-level dynamic binary translation system. This is achieved by separating of optimizing translation from execution flow into an independent thread which can then con-

currently share available processing resources with execution thread. Backgrounding is implemented whether by interleaving the two threads in case of a single-core (single processor) system or by moving optimization thread to an unemployed processor core in case of a dual-core (dual processor) system. In the first case the latency introduced to the system by the "heavy" phase of optimizing translation is reduced. In the second case, overlapping of execution and optimization threads also eliminates the time spent in dynamic optimization phase from the total run time of the original application under translation.

The specific contributions of this work are as follows:

- implementation of multithreaded infrastructure in a VM-level dynamic binary translation system;
- single processor system targeted implementation of background optimization technique where processor time sharing is implemented by interleaving optimizing translation with execution of original binary codes;
- dual processor system targeted implementation of background optimization technique where optimizing translation is being completely offloaded onto underutilized processor core.

The solutions described in the paper were implemented in the VM-level dynamic binary translation system LIntel, which provides full system-level binary compatibility with Intel IA-32 architecture on top of Elbrus architecture [9], [10] hardware.

## II. LINTEL

Elbrus is a VLIW (Very Long Instruction Word) microprocessor architecture. It has several special features including hardware support for full compatibility with IA-32 architecture on the basis of transparent dynamic binary translation.

LIntel is a dynamic binary translation system developed for high performance emulation of Intel IA-32 architecture system through dynamic translation of source IA-32 instructions into wide instructions of target Elbrus architecture (the two architectures are ISA-incompatible). It provides full system-level compatibility meaning that it is capable of translating the entire hierarchy of source architecture software (including BIOS, operating systems and applications) transparently for the end user (Fig. 1). As is noted above, LIntel is a co-designed system (developed along with the architecture, with hardware assistance in mind) and heavily utilizes all the features of architecture introduced to support efficient IA-32 compatibility.

In its general structure LIntel is similar to many other binary translation and optimization systems described before [11], [12], [13] and is very close to Transmeta's Code Morphing Software [14], [15]. As any other VM-level binary translation system, it has to solve the problem of efficient sharing of computational resources between translation and execution of original binary codes.

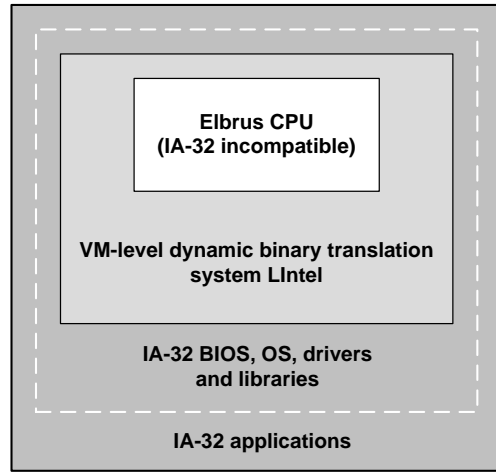


Fig. 1. VM-level dynamic binary translation system LIntel.

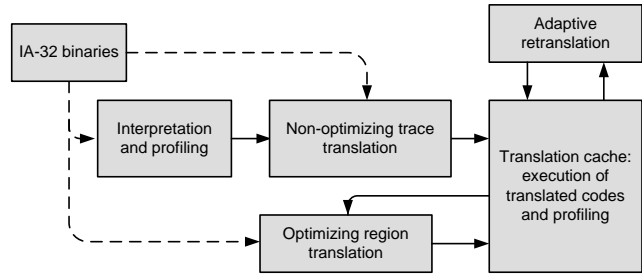


Fig. 2. Adaptive binary translation.

### A. Adaptive binary translation

LIntel follows adaptive, profile-directed model of translation and execution of binary codes (Fig. 2). It includes four levels of translation and optimization varying by the efficiency of the resulting Elbrus code and the overhead implied, namely: interpreter, non-optimizing translator of traces and two optimizing translators of regions. LIntel performs dynamic profiling to identify hot regions of source code and to apply reasonable level of optimization depending on executable codes behavior. Translation cache is employed to store and reuse generated translations throughout execution. Run-time support system controls the overall binary translation and execution process.

When the system starts, interpreter is used to carefully decode and execute IA-32 instructions sequentially, with attention to memory access ordering and precise exception handling. Non-optimizing translation is launched if execution counter of a particular basic block exceeds specified threshold.

Non-optimizing translator builds a *trace* which is a semantically accurate mapping of one or several contiguous basic blocks (following one path of control) into the target code. The building blocks for the trace are templates of the corresponding IA-32 instructions, where template is a manually scheduled sequence of Elbrus wide instructions. After code generation and additional corrections like actual constants and address values patching the trace is then stored into the translation

	Cycles per one source instruction translation	Translated code performance
Non-optimizing translation	1600	0.18
O0 optimization	30000	0.58
O1 optimization	1000000	1.0

Fig. 3. Average translation overhead per one IA-32 instruction and the performance of translated codes (normalized to O1).

cache. Trace translator produces native code without complex optimizations and focuses more on fast translation generation rather than code efficiency. It improves start-up time significantly as compared to interpretation. At the same time non-optimizing translation is only reasonable for executable codes with low repetition rate.

Traces are instrumented to profile hot code for O0-level optimizing translation. The unit of optimizing translation is a *region*. In contrast to traces, regions can combine basic blocks from multiple paths of control providing better opportunities for optimization and speculative execution (which is an important source of instruction level parallelism for VLIW processors).

O0-level translator is a fast region-based optimizer that performs basic optimizations implying low computation cost, including peephole, dead-code elimination, constant propagation, code motion, redundant load elimination, superblock if-conversion and scheduling.

Strong O1-level region-based optimizer is on the highest level of the system. The power of this level is comparable with high-level language optimizing compilers<sup>1</sup>. It applies advanced optimizations such as software pipelining, global scheduling, hyperblock if-conversion and many others, as well as utilizes all the architectural features introduced to support binary optimization and execution of optimized translations.

Region translations are stored in the translation cache as well. Profiling of regions for O1-level optimization is carried out by O0-level translations.

Optimized translations not always result in performance improvement. Unproven optimization time assumptions can cause execution penalty. These include incorrect speculative optimizations, memory mapped I/O access in optimized code (where I/O access is not guaranteed to be consistent due to memory operations merge and reordering), etc. Correctness of optimizations is controlled by the hardware at runtime. Upon detecting a failure, retranslation of the region is launched applying more conservative assumptions depending on failure type.

Fig. 3 compares average translation cost of one IA-32 instruction and the performance of translated codes for different levels of optimization. Adaptivity aims at choosing appropriate level of optimization throughout the translation and execution process to maintain overhead/performance balance.

Fig. 4 shows translation and execution time distribution for SPEC2000 tests running under Linux (operating system is

<sup>1</sup>In fact, O0/O1 notation of LIntel’s binary optimizers corresponds to conventional O2/O3-O4 optimization levels of language compilers.

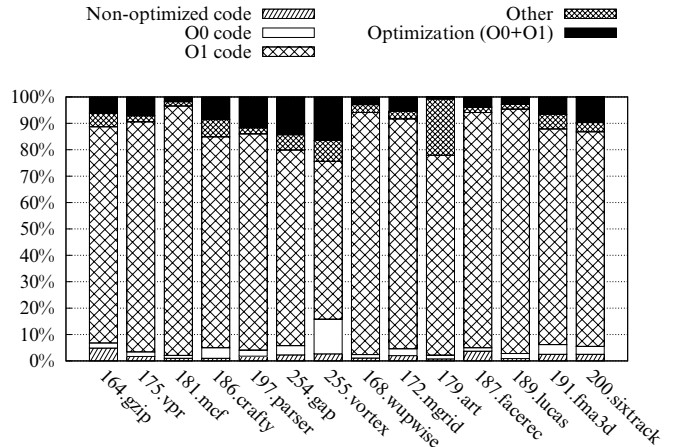


Fig. 4. Profile of binary translation in case of consecutive dynamic optimization.

being translated as well). While translated codes are executed most of the tests’ runtime, optimizing translation overhead is significant and equals to 7% on average.

### B. Asynchronous interrupts handling

One of the run-time support system functions is to handle incoming external (aka asynchronous) interrupts. The method of delayed interrupt handling allows to improve the performance of binary translated code execution and interrupt handling by specifying exactly where and when a pending interrupt can be handled. When interrupt occurs, interrupt handler only remembers this fact by setting corresponding bit in the processor state register and returns to execution. Interpreter checks for pending interrupts before next instruction execution. Due to efficiency reasons, non-optimized traces only include such checks in the beginning of basic blocks. Optimizing translators inject special instructions in particular places of a region code (where execution context is guaranteed to be consistent) that check for pending interrupts and force execution flow to leave region and switch to interrupt handler if needed.

This method of pending interrupt checks arrangement simplifies planning and scheduling of translated codes as there is no need to care about correct execution termination and context recovery at arbitrary moments of time. At the same time it allows LIntel to respond reactively enough to external events.

The bottleneck in this scenario is the presence of optimizing translation phase. If an interrupt occurs when optimization is in progress, it has to wait for optimization phase completion to be handled (Fig. 5). Due to computational complexity of optimizations employed, optimizing translation can consume significant amount of processor time and as such, the delay of response of the system to an external event can be noticeable (see evaluation in Section III-B).

## III. BACKGROUND OPTIMIZATION

To overcome the problems of performance overhead and latency caused by optimizing translation, the method of back-

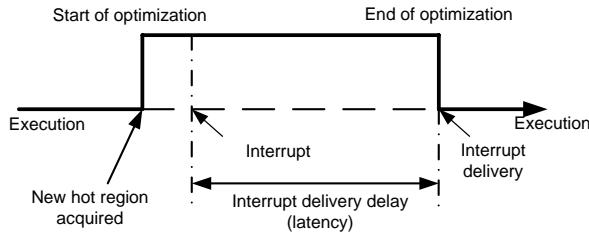


Fig. 5. Asynchronous interrupt delivery delay (latency) due to optimizing translation.

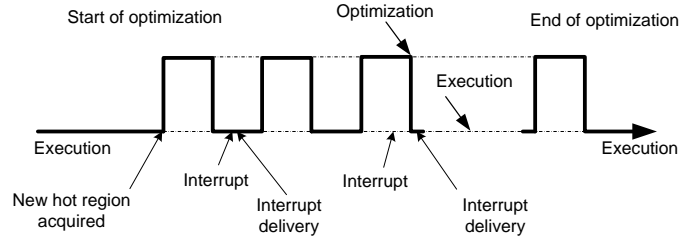


Fig. 6. Asynchronous interrupt delivery in case of interleaved background optimization.

ground optimization was employed in LIntel.

The concept of background optimization implies performing optimizing translation phase concurrently (or pseudo-concurrently) with the main binary translation flow of execution of original binary codes. Application-level binary translators usually implement this by utilizing native operating system’s multithreading interface and scheduling service to perform optimization in a separate thread. VM-level binary translation systems require internal implementation of multithreading to support background optimization.

In this section we describe implementation of background optimization in the VM-level DBTS LIntel. Two cases are considered: in the first case LIntel operates on top of a single-core target platform system; in the second case there are two cores available for utilization.

SPEC2000 tests are used to demonstrate the effect of background optimization implementation.

#### A. Execution and optimization threads

A multithreaded execution infrastructure was implemented in LIntel, with optimizing translation capable of running independently in a separate thread, which enabled execution and optimization threads concurrency. Execution thread activity includes the entire process of translation and execution of original binary codes, except for optimizing translation (of both O0 and O1 levels), i.e.: interpretation, non-optimizing translation, run-time support and execution itself. Optimizing translator is run in a separate optimization thread when new region of hot code is identified by the execution thread. When optimization phase completes, generated translation of the region is returned to the execution thread, which places it into the translation cache.

During the region optimization phase corresponding original codes are being executed either by interpretation or by previously translated codes of lower levels of optimization. Selection of new hot regions for optimization will not be launched unless current optimization activity completes.

By the end of optimization, memory pages that contain a source code of the region under optimization can get invalidated (due to DMA, self-modification, etc.). As such, before placing optimized translation of the region into the translation cache, execution thread must check region’s source code consistency and reject the region if verification fails. This routine is assisted by the memory protection monitoring

	Consecutive optimization	Interleaved (background) optimization
O1 phase mean time	1.54 s	3 s
O1 phase max time, $T_{O1\_max}$	8.8 s	29.5 s
interrupt delivery mean time with no optimization in progress	54 $\mu$ s	
interrupt delivery max time (with O1 phase in progress)	8.8 s ( $T_{O1\_max}$ )	1.7 ms

Fig. 7. Interrupt delivery time (CPU frequency = 300 MHz; thread time slice = 50000 cycles). O1-level optimization time is used as a reference as this phase consumes a greater number of processor cycles per source instruction as compared to O0-level optimization.

subsystem (introduced in the Elbrus hardware to support binary translation [16]) which controls source and translated (as well as translations-in-progress) codes coherency.

Separation of execution and optimization threads allows to schedule them across available processing resources in the same way as multitasking operating systems schedule processes and threads. By now, two simple strategies of processor time sharing were implemented in LIntel enabling optimization backgrounding for single-core and dual-core systems.

#### B. Background optimization in a single-core system

In case of a single-core system background optimization is implemented by interleaving of execution and optimization threads. Throughout optimizing translation of a hot region processor switches between the two threads. Scheduling is triggered by interrupts from internal binary translation dedicated timer “invisible” for executable codes under translation. Each thread is assigned a fixed time slice. When execution thread is active, incoming external interrupt has a chance to be handled without having to wait for region optimization to complete (Fig. 6). If there are no hot regions pending for optimization, execution thread fully utilizes the processor core.

To demonstrate single-core background optimization approach, a simple strategy of processor time sharing was chosen when both threads have equal priority, with equal time slices assigned (meaning that optimization thread’s processor utilization is 50%, in contrast to 100% utilization when optimizing consequently). As seen from Fig. 7, interleaving of execution and optimization improves interrupt delivery time significantly.

At the same time, as Fig. 8 demonstrates, this approach tend

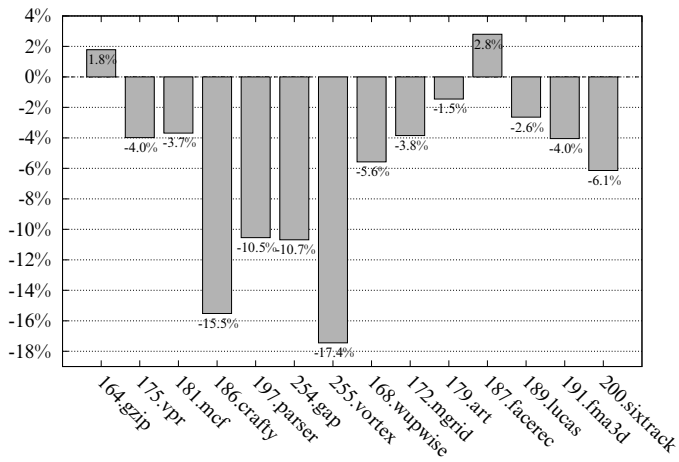


Fig. 8. Binary translation slow-down when interleaving optimization with execution (as compared to consecutive optimization).

to degrade binary translation performance.

Degradation can be explained by the fact that hot region optimization phase now lasts longer. As a result, optimized translations injection into execution is being delayed, meanwhile source binary codes are being executed non-optimized (or interpreted). Additional overhead comes with context switching routines.

Basically, single-core background optimization implementation is not of high priority currently. At the same time we believe that it is possible to improve its efficiency by tuning various parameters like execution and optimization threads' time slices and profiling thresholds to achieve earlier injection of optimized translations into execution process while keeping whole system latency acceptable. Besides, IA-32 "halt" instruction can be used as a hint to utilize free cycles and yield processor to optimization thread before the end of execution thread's time slice. Extensive study of execution and optimization threads' processor time utilization was made in [17].

### C. Background optimization in a dual-core system

In a dual-core system LIntel completely utilizes the second (unemployed otherwise) processor core to perform dynamic optimization in a background thread. In this case execution thread exclusively utilizes its own core and only interrupts execution to acquire next region for optimization and allocate generated translation when optimization completes.

As Fig. 10 demonstrates, overlapping of execution and optimization by moving optimization thread onto a separate core not only eliminates the problem of latency, but also increases overall binary translation system performance.

The resulting speed-up (6% on average) agrees good enough with dynamic optimization overhead estimated for the case of consecutive optimization (see Section II-A).

### D. Discussion and future works

As noted above, selection of hot regions in execution thread gets blocked unless optimization phase completes. However,

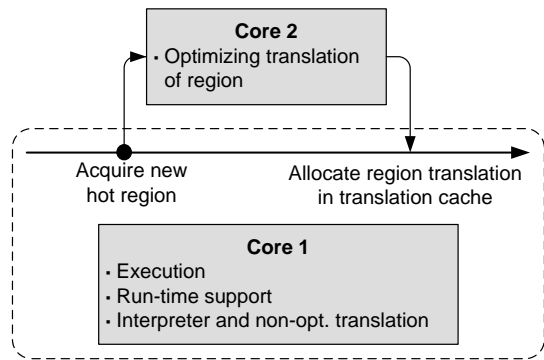


Fig. 9. Utilization of a separate processor core for dynamic optimization.

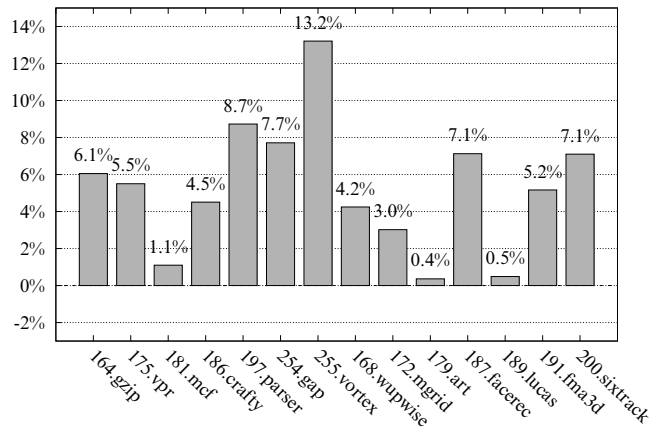


Fig. 10. Binary translation speed-up when optimizing on a separate processor core (as compared to consecutive optimization).

profile counters continue to grow, and by the end of optimization there may be several nonoverlapping regions in the profile graph with counters exceeding threshold. As counters are checked during execution of corresponding translated codes, next optimizing translation will be launched for the first region executed. Not necessarily will this region be the hottest one. As such, a problem of suboptimal hot region selection arises which also needs to be addressed (profile graph traversal can be quite time-consuming and is not an option).

The profile of binary translation for SPEC2000 tests (Fig. 4) suggests that current optimization workload is not enough to fully utilize optimization thread affiliated processor core, which will run idle most of the application run time. To improve its utilization ratio, optimizing translator can be forced to activate more often. This can be achieved by dynamically decreasing of hot region profiling threshold depending on current load of the core affiliated with optimizing translator. When execution activity is naturally low, this core should be halted due to energy efficiency reasons.

This is reasonable to ask why not utilize unemployed processor core to execute source binary codes. In other words, if there are more than one target architecture microprocessor core in the system, source architecture system software (e.g.

operating system) could "see" and utilize the same number of cores. Current Elbrus architecture implementation (used in this paper) does not satisfy IA-32 architecture requirements concerning organization of multiprocessor systems. As a result, IA-32 multiprocessor support is not possible on top of Elbrus hardware. But we hope to implement this scenario in the future. Still, we believe that having processor cores solely utilized for dynamic optimization is reasonable due to a following:

- different classes of software (legacy software, software for embedded systems, etc.), not always developed with multiprocessing or multithreading in mind, can benefit from multicore or multiprocessor systems when being executed through binary translation with background optimization option;
- keeping in mind the tendency towards ever increasing number of cores per chip, it seems reasonable to utilize some cores to improve dynamic binary translation system performance; not only optimizing translator can consume this resources; other jobs that could also be performed asynchronously include identification and selection of code regions for optimization [18], software code prefetching [19], persistent translated code storage access [20]<sup>2</sup>, etc.

Finally, we think that a promising direction for future research and development is building a binary translation infrastructure that could support unrestricted number of execution (in terms of source architecture virtual machine; so that operating system under translation could "see" more than one processor core), optimization and other threads and schedule them efficiently across the available computational resources depending on their quantity, load and binary codes execution behavior.

#### IV. CONCLUSION

The paper addresses the problem of optimization overhead in dynamic binary translation systems and presents the application of background optimization technique in full system dynamic binary translator LIntel. Implementations for single-core and dual-core systems are considered. In the first case backgrounding is implemented by interleaving execution and optimization, while in the second case dynamic optimization is completely moved onto a separate processor core. In both cases background optimization solves the problem of high latency caused by dynamic optimization which is particularly important for full system execution environment. Performing optimization on a separate core also eliminates optimization overhead from the application run time thus improving binary translation system performance in general.

<sup>2</sup>Asynchronous access to a persistent code storage (aka CodeBase) has already been implemented in LIntel by the moment but is not covered in this paper as we only consider the effect of background optimization implementation.

#### REFERENCES

- [1] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.
- [2] S. Campanoni, G. Agosta, and S. C. Raghizzi, "ILDJIT: a parallel dynamic compiler," in *VLSI-SoC'08: Proceedings of the 16th IFIP/IEEE International Conference on Very Large Scale Integration*, 2008, pp. 13–15.
- [3] C. J. Krintz, D. Grove, V. Sarkar, and B. Calder, "Reducing the overhead of dynamic compilation," *Software: Practice and Experience*, vol. Volume 31 Issue 8, pp. 717–738, 2001.
- [4] J. Mars, "Satellite optimization: The offloading of software dynamic optimization on multicore systems (poster)," in *PLDI '07: 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.
- [5] P. Unnikrishnan, M. Kandemir, and F. Li, "Reducing dynamic compilation overhead by overlapping compilation and execution," in *Proceedings of the 11th South Pacific Design Automation Conference (ASP-DAC '06)*. Piscataway, NJ, USA: IEEE Press, January 2006, pp. 929–934.
- [6] M. J. Voss and R. Eigenmann, "A framework for remote dynamic program optimization," in *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, 2000, pp. 32 – 40.
- [7] W. Zhang, B. Calder, and D. M. Tullsen, "An event-driven multithreaded dynamic optimization framework," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 87–98.
- [8] H. Guan, B. Liu, T. Li, and A. Liang, "Multithreaded optimizing technique for dynamic binary translator CrossBit," *Computer Science and Software Engineering, International Conference on*, vol. 5, pp. 945–952, 2008.
- [9] B. Babayan, "E2k technology and implementation," in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 2000, pp. 18–21.
- [10] V. Volkonskiy, "Optimizing compilers for Elbrus-2000 (E2k) architecture," in *4th Workshop on EPIC Architectures and Compiler Technology*, 2005.
- [11] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach., "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 191.
- [12] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: A profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, 1998.
- [13] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *Computer*, vol. 33, no. 3, pp. 54–59, 2000.
- [14] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2003.
- [15] A. Klaiber, "The technology behind Crusoe processors," Transmeta Corporation, Tech. Rep., January 2000.
- [16] A. V. Ermolovich, "Methods of hardware assisted dynamic binary translation systems performance improvement," Ph.D. dissertation, Institute of microprocessor computing systems, Moscow, 2003.
- [17] P. Kulkarni, M. Arnold, and M. Hind, "Dynamic compilation: the benefits of early investing," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2007, pp. 94–104.
- [18] J. Mars and M. L. Soffa, "MATS: Multicore adaptive trace selection," in *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems (STMCS 2008)*, April 2008.
- [19] J. Mars, D. Williams, D. Upton, S. Ghosh, and K. Hazelwood, "A reactive unobtrusive prefetcher for multicore and manycore architectures," in *Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms (SHCMP)*, June 2008.
- [20] A. V. Ermolovich, "CodeBase: persistent code storage for dynamic binary translation system performance improvement," *Information technologies*, vol. 9, pp. 14–22, 2003.

# The ARTCP header structure, computation and processing in the network subsystem of Linux kernel.

Anatoliy Sivov  
Yaroslavl State University  
Yaroslavl, Russia  
mm05@mail.ru

V.A. Sokolov (research supervisor)  
Yaroslavl State University  
Yaroslavl, Russia

*Abstract*—ARTCP is a transport level communication protocol based on TCP. It uses temporal characteristics of data flow to control it, that allows to split algorithms of congestion avoidance and reliable delivery. The article discusses possible ARTCP header structure and practical aspects of forming the header and calculation of the header fields. It demonstrates the possibility of transparent replacement of TCP with ARTCP due to flexible ARTCP connection setup implementation and ARTCP packets structure compatibility with TCP. The questions of precise time dispatching of the received packets are discussed. The Linux kernel interfaces for time measurement are described as well as the clock source abstraction layer and its implementation.

*Networking; transport protocol; ARTCP; time measurement; Linux kernel.*

## I. INTRODUCTION

TCP is the most widespread transport protocol with the reliable data delivery today. It has become the industry standard de facto. However, this protocol is not ideal. It has at least two big disadvantages. The first of them is that its data flow management algorithm results in periodic network congestion by design. It leads to unnecessary packet loss and latency increase. The second is inefficient bandwidth use in the unreliable physical environment (i.e. wireless networks) where BER can be reasonably high. This disadvantage comes from TCP impossibility to distinguish packet loss due to congestion and packet loss due to some transmission errors.

Adaptive Rate TCP (ARTCP) is a transport protocol with the reliable data delivery that uses some TCP principles but tends to solve these two TCP's disadvantages. ARTCP uses temporal characteristics of data flow in its data flow management algorithm. It allows ARTCP to determine bandwidth efficiently without need in periodic network congestion. The main feature of ARTCP is a logical separation of error correction and data flow management. Due to this separation ARTCP is able to use bandwidth more efficiently than TCP in unreliable environment [1]. The ideas of ARTCP and protocol's description can be found in [1] and [2].

This article is a research-in-progress report about ARTCP implementation in Linux kernel. It considers the following things: transparent replacement of TCP with ARTCP, coexistence of TCP and ARTCP, ARTCP header structure and

processing, and computation of ARTCP header fields values in Linux kernel.

The Linux kernel is chosen as the target platform of ARTCP implementation for several reasons:

- Linux is an open source system. It allows to use its source code and modify it. So, Linux allows the most flexible implementation process it could be what is very important while implementing protocol that is closely associated with the existing one. It makes implementation to be much more easy and efficient comparing with writing the module for operating system with closed sources.
- Linux has very good network subsystem. When implementing transport protocol it is very desirable to have good implementation of underlying protocols as well as networking implementation at whole. Linux has advantages of very good networking stack with well-implemented layers abstraction and object-oriented socket concept. The other advantage is a TCP implementation modularity.
- Linux is a popular, industry-choice operating system. Linux is the most popular operating system in the industry comparing with the other open source operating systems. The wish to create an implementation of ARTCP for OS widely used in the industry influenced on the choice among variety of open source operating systems.
- Linux can run on various hardware platforms. Linux has support of many hardware platforms including x86, ia64, x64, arm, avr, mips, ppc and so on. ARTCP implementation written hardware-independently will be supported on all of these platforms.

## II. ARTCP AND TCP

It is very important to have TCP working on the system which supports ARTCP. Taking into account that ARTCP is considered as transparent replacement of TCP the responsibility for this lies on ARTCP implementation. It is so because applications (or overlying protocols such as HTTP) does not know whether they establish TCP or ARTCP connection unlike the situation when they use the other

transport protocols (UDP or something more exotic like SCTP).

We have chosen to support TCP on ARTCP-featured systems in two ways. At first, operating system administrator must have a simple capability to choose whether to use TCP or ARTCP. This problem is solved with addition of `tcp_enable_artcp` kernel parameter with possible values 0 or 1 where 0 means that ARTCP is disabled (and TCP is used) and 1 means that ARTCP is enabled (and ARTCP is used if possible for all connections). Like any other kernel parameter this parameter is accessible in runtime with `/proc/sys/` interface. Its value can be changed with writing 0 or 1 to `/proc/sys/net/ipv4/tcp_enable_artcp` file. Also its value can be set in `/etc/sysctl.conf` file in the same way as any other kernel parameter.

Secondly, ARTCP implementation must be able to fall back to TCP if the other end of connection does not support ARTCP. This capability allows to have ARTCP enabled in the TCP world. It is very useful to be able to switch to TCP in the kernel without any packet retransmission or disturbing overlying protocol or an application that uses ARTCP/TCP connection but imposes ARTCP packet structure and connection setup to be TCP-compatible. This article suggests ARTCP header structure and connection setup that are TCP-compatible and makes it possible to fallback to TCP at any time of connection. Also it represents implementation of ARTCP header processing in Linux networking subsystem and discusses the questions of ARTCP header fields values calculation.

### III. ARTCP HEADER STRUCTURE

As mentioned above ARTCP header structure must be TCP-compatible. Let us consider TCP header and how it is possible to extend it to fit ARTCP needs.

TABLE I. TCP HEADER

Bit	0-3	4-7	8-15	16-31
0	Source port		Destination port	
32	Sequence number			
64	Acknowledgment number			
96	Data offset	Reserved	Flags	Window size
128	Checksum		Urgent data pointer	
160	Options (optional field)			

TCP has a native support to extend header called TCP options. TCP options are considered in [3] and succeeding RFCs and have the following format: first byte contains option number, second byte contains option length (in bytes including 2 bytes for number and length fields) and 0 or more bytes (specified in the length field) contains option value. There are only two exception for this format. Option number 0 is one byte long. It is used to mark the end of options list. The other exception is an option number 1 which is used for padding to

align other options on 32-bit boundaries. TCP header allows up to 40 bytes to be used for options list.

ARTCP requires only two extra fields for its functionality: PS field and TI field. Each one of them can be represented as 32 bit number. PS (Packet Sequence) field holds unique packet sequence number (modulo  $2^{32}$ , of course). According to [1] this field must be presented in every ARTCP packet with payload data. ARTCP receiver uses this field to determine whether the packet received is the next packet in the stream comparing with the previous received packet. So, the value of PS field in ARTCP is to help receiver to distinguish the packet in order sent first time from the packet in order sent again (due to some packet loss). Indeed data presented in TCP header are enough to distinguish segments in order from segments out of order using the sequence number field. However, in terms of TCP there is no difference between the segment sent first time and the segment sent again (due to retransmission) because both of them share the same sequence number and there is no any indication of retransmitted segment in TCP.

The second field, TI (Time Interval) is used by ARTCP to compute the value of stream's duty ratio. The paper [4] suggests to carry the time interval measured between two consecutive moments of ARTCP packets arrival. It is considered more useful to carry time intervals in the TI field instead of carrying duty factor what was suggested in [1]. ARTCP receiver puts this field in every its acknowledgment packet (packet that has ACK flag). It is necessary to use real ("human") time units for time interval resolution in TI that must not depend on hardware used by sender or receiver (i.e. these units must not be CPU ticks or something like that). The paper [4] suggests to use microseconds for this purpose. Time measurement with this resolution is possible on the most of hardware used nowadays and has a sufficient precision for the existing problem. TI field may be represented with 32 bit number.

Both TI and PS can take the form of TCP options in ARTCP header. In this case they must take at least 6 bytes (8 bytes to keep the 32 bit alignment) – 1 byte for option number, 1 byte for option length and 4 bytes for option value (and 2 bytes for alignment). For today implementation PS field may use option number 253 and TI field may use option number 254. These option numbers are chosen in conformity with RFC 4727 [5] to use in experiments. They must be changed later in conformity with RFC 2780 [6]. The use of TCP options numbers 253 and 254 is regulated in RFC 3692 [7].

Summing up, ARTCP header is a valid TCP header extended with two TCP options called PS and TI. Every ARTCP packet with payload data as well as packet with SYN or FIN flag contains PS field with packet sequence number in the header. This number is one more than the number of previous segment transmitted by the sender (excluding case of retransmission during connection setup). Namely, if the segment with value N in the field PS was transmitted by sender but was not delivered (or its acknowledgment was not received by sender) then this segment is retransmitted (with possible repacketization) but has value N+1 (modulo  $2^{32}$ ) in the PS field. Every ARTCP packet with ACK flag must contain TI field in its header. TI field must have 0 as value if this packet is



an acknowledgment for a packet that contains in PS field value that differs from the value of previous received packet incremented by one modulo  $2^{32}$ . Otherwise, field TI must contain calculated value of the time interval. Both fields PS and TI are written in the network order.

#### IV. ARTCP HEADER PROCESSING

ARTCP shares a lot of algorithms with TCP, also ARTCP implementation must allow fallback to TCP if one of connection ends does not support ARTCP. So it's decided to use existing network subsystem of Linux kernel, implementation of Ipv4/TCP stack in particular, to implement ARTCP.

As described above, ARTCP header differs from TCP header with presence of PS and TI fields only. These fields are represented in the form of TCP options so that it is necessary to modify the code that implements TCP options reading and writing to implement reading and writing of ARTCP header.

To form TCP options to be sent in the header Linux kernel uses **struct tcp\_out\_options**, the structure that contains the fields with values of different TCP options supported by network stack of Linux and bit field **options** to set flags that indicates which options must be written to the header of the current TCP packet. To implement writing of ARTCP header the fields for TI and PS values were added to this structure, also bit flags, that indicates the presence of the fields and are used in options bit field, were created.

For this structure to be filled correctly the functions that initializes the instance of this structure were modified. These functions are **tcp\_syn\_options**, **tcp\_synack\_options** and **tcp\_established\_options**. They forms options for SYN packets, SYN-ACK packets and the other packets, respectively. The modified functions checks whether socket is in the ARTCP mode and if so adds information about needed PS or/and TI field.

To write TCP options into network buffer that contains the header **tcp\_options\_write** function is used. This function is modified as well to write PS and TI fields to the header if it is specified in the instance of the modified **struct tcp\_out\_options**. All these modifications make it possible to form ARTCP packets in the Linux network subsystem.

To parse the options of the TCP header in the packet received Linux calls **tcp\_parse\_options** function, which analyzes the received data and forms the instance of **struct tcp\_options\_received** by writing received values into it. To support ARTCP this structure was extended with fields **ps** and **ti** that contains values of fields PS and TI of the received ARTCP packet and bit field **artcp\_options** that determines which ARTCP fields were actually presented in the header (PS, TI, both or none). Also **tcp\_parse\_options** function must be modified to handle ARTCP fields and form the modified structure.

To handle ARTCP packets properly it is necessary to process received ARTCP fields depending on the connection state and the presence/absence of payload data in the received packet. In Ipv4/TCP stack received SYN packet is processed in

**tcp\_v4\_conn\_request** function. The modified code of this function checks the kernel parameters set by administrator by reading **sysctl\_tcp\_enable\_artcp** variable which has information whether ARTCP is globally enabled in the system or not. If ARTCP is enabled **tcp\_v4\_conn\_request** checks the presence of field PS in the received SYN packet and the correctness of its value as well as the absence of field TI. If all checks are passed then function puts socket in ARTCP mode and initializes all resources needed by ARTCP connection. Otherwise, the function puts socket in TCP mode.

If the socket has already sent SYN packet (and is in SYN-SENT state) then the packets received with this socket are handled with **tcp\_rcv\_synsent\_state\_process**. The modified code of this function checks the presence of PS and TI fields in the header and correctness of their values when SYN-ACK packet is received for socket in ARTCP mode. If checks are not passed then the function puts the socket in TCP mode. Otherwise, function initializes all resources needed by ARTCP connection.

For established ARTCP connection all received packets are handled with **tcp\_rcv\_established** function. The modified code of this function processes ARTCP packets for the socket in ARTCP mode. If the header of the received packet has no needed fields PS (for packet that has payload data) or TI (for packet that has ACK) or the header of the packet without payload data has field PS then socket falls back to TCP. If ACK packet received then the value of field TI is passed to data flow management algorithm of ARTCP. For packet that has payload data the function checks the value of field PS. If this packet is the next (after previous received packet) packet sent by the other end according to this field then it is necessary to calculate the difference between the time of arrival of these two packets to send it in the field TI of the acknowledgment packet. Otherwise, the ACK packet will contain 0 in the field TI.

#### V. TIME MEASUREMENT FOR TI FIELD IN LINUX

TI requires time measurement with microseconds resolution what may be nontrivial problem. Linux kernel guarantees the availability of so-called "system clock" represented with **jiffies** interface. **Jiffies** can be considered as read only global variable which is updated with **HZ** frequency. **HZ** is a compile-time kernel parameter whose reasonable range is from 100 to 1000 Hz [8]. So, it is guaranteed to have an interface for time measurement with 1-10 milliseconds resolution.

The availability of more precise techniques for time intervals measurement is hardware-dependent. Let us consider x86 architecture as an instance. All IMB-compatible PCs have Programmable Interval Timer (PIT) known as chip Intel 8523 (or Intel 8524 and other analogues). This chip (or an analogue, i.e. south bridge of the motherboard may have this functionality) has three independent 16-bit counters called channels. Channel 0 usually is used for clock interrupts generation. Channel 1 assists in generating timing for DRAM memory refreshes. And channel 2 commonly generates PC speaker tones. PIT allows to achieve 1 ms time resolution.

The other clock source is Real Time Clock (RTC). RTCs usually have an alternate source of power, so they can continue to keep time while the primary source of power is off or unavailable. RTC's functionality is provided with south bridge in the modern motherboards. However, RTC allows time measurement with 1 ms resolution.

The most modern x86 motherboards have Advanced Programmable Interrupt Controller (APIC) and APIC timer as a consequence. This timer's frequency equals CPU bus frequency what allows time measurement with a high resolution (about 10 nanoseconds). The other benefit is that in contradistinction to PIT and RTC local APIC timer does not require call to I/O port. The most uniprocessor PCs above Pentium 4 explicitly prohibits APIC by disabling it in BIOS.

Also systems, that have a support of Advanced Configuration and Power Interface (ACPI), have so-called Power Management timer (PM timer). Unlike APIC timer, it is possible with PM timer to have a reliable time independently on CPU speed changes due to active power management with OS.

At the beginning of 2000s Intel and Microsoft corporations has developed High Precision Event Timer (HPET) [9]. This timer has a high frequency (not less than 10 MHz) and uses 64-bit counter. Often it is a most preferable high-precision clock source in the system.

In addition to peripheral timers x86 computers have on-chip (on-CPU) 64-bit counter called Time Stamp Counter (TSC). Comparing with the other counters this one has advantages of less read latency and high resolution. The frequency of this counter on different CPUs varies and can equal CPU frequency or CPU bus frequency. There are two major problems to use this counter as clock source. The first one is that Time Stamp Counters may be not synchronized between cores of SMP [10]. The second is that frequency of TSC may be non-constant (due to power management or processor frequency changes on idle and so on).

Intel's software developer's manual [11] describes in depth the differences in TSC implementation on different Intel CPU families. It also describes the way to recognize whether CPU has TSC with invariant rate. Most AMD processors have TSC that is unusable as a reliable clock source because of certain circumstances.

The facilities for time interval measurement in x86 architecture listed above give an idea of the difficulty to solve this problem more precisely for different processors. The support of other hardware architectures (arm, mips and so on) highly increases this difficulty. The other problem is a non-triviality of time units translation from "machine" time units used in the chosen device to "human" time units (for example, microseconds needed by ARTCP). Reading the report [12] shown in 2005 in Ottawa (Canada) at a symposium devoted to Linux you can get an idea on the complexities associated with the solution of this problem.

Fortunately, Linux kernel provides the means for solving these problems. To have a possibility to use different hardware counters and timers "clock source" concept is implemented in

Linux kernel. According to this concept, each hardware architecture supported by Linux implements for each available facility a "clock source" interface. It does it by initializing an instance of **struct clocksource** interface and registering it in operating system with call to either **clocksource\_register\_khz** or **clocksource\_register\_hz**. **Struct clocksource** has field **rating** that allows Linux to choose the best "clock source" available for the specified hardware. Best "clock source" corresponds to the registered instance of **struct clocksource** with the biggest value of field **rating**. The values of this field are logically interpreted in that way: 1-99 – unfit for real use (only available for bootup and testing purposes), 100-199 – base level usability (functional for real use, but not desired), 200-299 – good (a correct and usable clocksource), 300-399 – desired (a reasonably fast and accurate clocksource), 400-499 – perfect (the ideal clocksource, that is a must-use where available).

Since the best "clock source" has been chosen Linux kernel is able to read its counter values by calling the function passed in field **read** of **struct clocksource**. This function returns the value in abstract "machine" time units represented with **cycle\_t** data type. Linux kernel can use the values of field **mult** and field **shift** of **struct clocksource** to translate this value to nanoseconds.

Linux kernel provides various interfaces for indirect work with "clock sources" and to retrieve the values of time in "human" units. The most interesting of them are **getnstimeofday** and **getrawmonotonic** functions. Both of these functions return the value of time as an instance of **struct timespec**. This structure consists of two fields: **tv\_sec** that carries seconds and **tv\_nsec** that carries nanoseconds. The most significant difference between these functions is that the former unlike the second uses NTP correction to adjust value it returns. The absence of this correction in **getrawmonotonic** allows to use this function to compute time intervals where the correspondence of the time set on the machine to the actual time does not matter. Based on these considerations, ARTCP implementation uses **getrawmonotonic** interface to calculate time intervals between two consecutive received ARTCP packets with payload data.

The first time reading with **getrawmonotonic** happens in **artcp\_init** function in the process of ARTCP connection initialization when either socket, that sent ARTCP packet with ACK, receives ARTCP packet with SYN-ACK, or socket receives ARTCP packet with SYN (and ARTCP is globally enabled). Subsequent readings occur when ARTCP packets with payload data are received. Moreover, if the value of the field PS in the received packet is one greater (modulo  $2^{32}$ ), than the value of the field PS in the previous received packet, then the value for the field TI of the acknowledgment packet is calculated with call to **artcp\_ts\_diff\_to\_ti** function. The function **artcp\_ts\_diff\_to\_ti** takes two **struct timespec** arguments, that represent the time interval, and returns the difference between these moments of time in microseconds.

Summarizing the material described in this article, we can conclude that, having made the above changes in the source code of Linux network subsystem, we get the implementation of ARTCP packets processing (receiving, sending, calculation

of field values) that is completely independent from the implementation details of data flow management function and the other parts of ARTCP. Moreover, the implementation is cross-platform (in the hardware) and uses the best available hardware facility for time intervals calculation with the ability to increase the resolution of the field TI up to nanoseconds. It is also worth nothing that the implementation does not conflict with the existing functionality of Linux network subsystem, allowing the latter to use TCP connections simultaneously with ARTCP and even switch ARTCP connections to TCP mode.

- [1] I. V. Alekseev, V. A. Sokolov, D.U. Chaly. Modeling and analysis of Transport protocols for computer networks. Yaroslavl State University, 2004. (in Russian)
- [2] I. V. Alekseev, V. A. Sokolov. Compensation Mechanism for Adaptive Rate TCP. // 1-St International IEEE/Popov Seminar "Internet: Technologies A and Services". P. 68-75, October 1999
- [3] J. Postel. Transmission Control Protocol. // RFC 793 (STD7). 1981.
- [4] I. V. Alekseev, S. A. Merkulov, A. A. Sivov. "Aspects of practical implementation of ARTCP in Linux kernel 2.6" // Modeling and analysis of information systems. Volume 17, №2. Yaroslavl: Yaroslavl state university, 2010. P. 144-149 (in Russian)
- [5] B. Fenner. Experimental Values in IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers. // RFC 4727. 2006.
- [6] S. Bradner, V. Paxson. IANA Allocation Guidelines For Values In the Internet Protocol and Related Headers. // RFC 2780. 2000.
- [7] T. Narten. Assigning Experimental and Testing Numbers Considered Useful. // RFC 3692. 2004.
- [8] J. Corbet, A. Rubini, G. Kroah-Hartman. Linux Device Drivers. 3rd edition. O'Reilly, 2005.
- [9] IA-PC HPET (High Precision Event Timers) Specification. Rev. 1.0a. Intel Corporation, 2004.
- [10] AMD Technical Bulletin – TSC Dual-Core Issue & Utility Fix. Advanced Micro Devices, Inc. 2007.
- [11] Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3A: System Programming Guide, Part 1. Intel Corporation. January 2011.
- [12] J. Stultz, N. Aravamudan, D. Hart. We Are Not Getting Any Younger: A New Approach to Time and Timers. // Proceedings of the Linux Symposium. Vol. 1. P. 219-232, July 2005.

# A new double sorting-based node splitting algorithm for R-tree

Alexander Korotkov  
National Research Nuclear University MEPhI  
31 Kashirskoe shosse  
Moscow, Russian Federation  
Email: aekorotkov@gmail.com

**Abstract**—A storing of spatial data and processing of spatial queries are important tasks for modern databases. The execution efficiency of spatial query depends on underlying index structure. R-tree is a well-known spatial index structure. Currently there exist various versions of R-tree, and one of the most common variations between them is node splitting algorithm. The problem of node splitting in one-dimensional R-tree may seem to be too trivial to be considered separately. One-dimensional intervals can be split on the base of their sorting. Some of the node splitting algorithms for R-tree with two or more dimensions comprise one-dimensional split as their part. However, under detailed consideration, existing algorithms for one-dimensional split do not perform ideally in some complicated cases. This paper introduces a novel one-dimensional node splitting algorithm based on two sortings that can handle such complicated cases better. Also this paper introduces node splitting algorithm for R-tree with two or more dimensions that is based on the one-dimensional algorithm mentioned above. The tests show significantly better behavior of the proposed algorithms in the case of highly overlapping data.

## I. INTRODUCTION

Spatial data processing is an important task for modern databases. Since the volume of information in databases increases continuously, the database management systems (DBMS) need spatial index structures in order to handle spatial queries efficiently. The problem of spatial indexes is that there is no ordering which reflects proximity of spatial objects [5]. This is why B-tree [3] can not handle spatial object efficiently. R-tree [7] is the most well-known index structure for spatial data. R-tree is a height balanced tree like B-tree, which hierarchically splits space into possibly overlapping subspaces. Spatial objects in R-tree are approximated by minimal bounding rectangles (MBRs), see figure 1. Leaf node entry of R-tree contains MBR of spatial object and a reference to the corresponding database object. An entry of non-leaf node of R-tree contains reference to the child node and MBR of all rectangles in child node. Since the rectangles of a same node of R-tree can overlap, exact match query may lead to multipath tree scan. This forms significant difference of R-tree from such data structures as B-tree. The number of query paths and, in turn, the number of node accesses of non-exact match query also strongly depends on degree of rectangle overlap. R-tree was originally designed for access to multidimensional data, but it is also applied for one-dimensional intervals [10].

The quality of R-tree strongly depends on the node splitting algorithm. The task of node splitting is to split entries of the

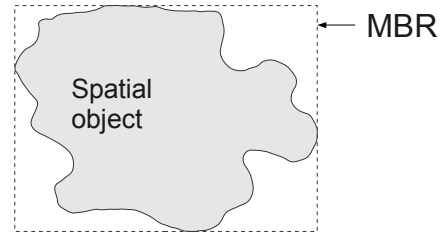


Fig. 1. MBR illustration

overflowed node into two groups which will form two new nodes. Node splitting algorithm substantially determines the area and degree of overlap of the tree rectangles. In turn these parameters determine the probability of multipath queries. The following parameters can be used in order to estimate the quality of a node splitting:

- The overlap of bounding rectangles. The smaller overlap of entry rectangles leads to the smaller probability of multipath queries.
- The coverage of bounding rectangles. The coverage of a split is a total area of bounding rectangles. In general smaller coverage leads to the smaller probability of multipath queries when query area is relatively large [1].
- Storage utilization. As the measure of storage utilization, a ratio between a numbers of entries in the smaller group and the greater group can be used. Typically, constraint is imposed on this parameter, i.e., the minimal number of entries in the resulting node  $m$  is defined. Restriction of this parameter is very reasonable, but this parameter can also be an optimization target. The higher ratio leads to the smaller tree balancing during construction. In turn, this influences the tree quality.

The illustration of dilemma between less overlap and less coverage is given on figure 2.

The paper is organized as follows. Section II describes node splitting algorithms which currently exist. Section III introduces double sorting-based one-dimensional node splitting algorithm and its generalization for multidimensional case. Section IV provides the experimental comparison of the proposed algorithm with other existing algorithms. Section V is a conclusion.

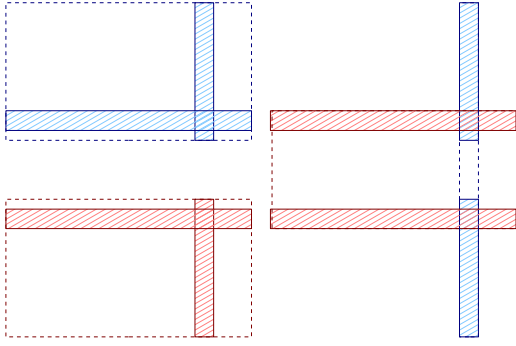


Fig. 2. Illustration of overlap vs. coverage dilemma

## II. RELATED WORK

Originally Guttman in [7] introduced three node splitting algorithms:

- Exponential algorithm. This algorithm searches for global minimum of the area covered by rectangles by the enumerations of all possible splits. This method is too CPU expensive, because it requires exponential time.
- Quadratic algorithm. This algorithm consists of two steps. At the first step, two seeds of two resulting groups are selected. The seeds are selected as the rectangles that have maximal difference between their MBR area and their own area. At the second step, all other rectangles sequentially join some of the groups. Each time the rectangle for which the increase of MBR area due to its joining to one of the groups is maximal joins the group which MBR area increases less.
- Linear algorithm. This algorithm is similar to quadratic one, but it has two differences that make it linear. At first, seeds are selected along the axis that allows avoiding comparison of each pair of rectangles. The second is that rectangles join the groups in arbitrary order.

In [6] Green's algorithm was proposed. This algorithm is similar to Guttman's linear algorithm, but it uses sorting along the chosen axis and splitting entries at halves between the groups according to the sorting.

In [4] R\*-tree splitting algorithm was proposed. This work contains tree construction modifications as well as new node splitting algorithm. The important feature of this work is using rectangle margin as an optimization criterion of node splitting. This algorithm is similar to Green's algorithm, but has two differences. At first, it chooses axis for splitting that minimizes the sum of margins of MBR groups among all possible sorting-based splits along this axis. At second, it does not split entries at halves, but finds the minimal overlap between all splits based on sorting along this axis. In [12] the comprehensive performance analysis of R\*-tree is presented. The optimization of R\*-tree for non-uniform data is presented in [9].

In [2] a new linear algorithm was proposed. This algorithm makes splits of rectangles along axes based on the closeness

of rectangles to value boundaries of the axes. After that, the choice is made among the splits by comparison of the overlaps and distribution ratios.

Since applications of R-tree exist for one-dimensional case, one-dimensional split for R-tree can be considered as a separate problem. One of the negative aspects of R-tree application to one-dimensional case is weak performance of high-overlapping data, such as validity interval or transactional time intervals [11]. This aspect can be partially eliminated by introducing new node splitting algorithm for one-dimensional case which deals better with highly overlapping data.

Guttman's quadratic and linear algorithms can be easily applied to one-dimensional case. For Guttman's quadratic algorithm there is no matter to use quadratic algorithm for picking seeds, because most distant seeds can be found as the intervals which contain the general lower and upper bound, correspondingly. Green's and R\*-tree splitting algorithms comprise one-dimensional split as their part. A new linear algorithm also can be applied to one-dimensional case, but we have only one axis for split and will not have to choose among the axes.

## III. PROPOSED ALGORITHM

### A. Definitions

In one-dimensional splitting algorithm, the input entries contain a set  $I$  of the intervals  $x_i: I = \{x_i\}$ . An interval is the pair of the lower and the upper bounds:  $x_i = (l_i, u_i)$ . The general lower bound is  $l = \min\{l_i\}$ , and the general upper bound is  $u = \max\{u_i\}$ . At first, the consideration of splits will be limited by the splits in which one group contains general lower bound and another group contains general upper bound. For this class of splits we will say that a pair  $(a, b)$  is a splitting pair, if any interval from  $I$  is bounded by  $(l, a)$  or  $(b, u)$ :  $\forall x(x \in I \Rightarrow (x \subseteq (l, a)) \wedge (x \subseteq (b, u)))$ . In other words,  $a$  and  $b$  are the upper and the lower bound of groups, respectively, for some split of split class under consideration. Let us note that sometimes the splits which are not contained in this class of splits are reasonable. In the figure 3, a split of this class is shown. In the figure 4, a split for the same dataset is shown. In that split, one group stretches from the general lower bound to the general upper bound while another group has rather small area. This split can not be produced by splitting pair.

We will say that the split pair  $(a, b)$  is a corner splitting pair if  $(a \in \{u_i\}) \wedge (b \in \{l_i\}) \wedge ((\forall t(t < a \Rightarrow \exists x(x \in I \Rightarrow (x \not\subseteq (l, t)) \wedge (x \not\subseteq (b, u)))) \vee (\forall t(t > b \Rightarrow \exists x(x \in I \Rightarrow (x \not\subseteq (l, a)) \wedge (x \not\subseteq (t, u))))))$ . In other words,  $a$  is one of the upper interval bounds,  $b$  is one of the lower interval bounds, and  $a$  can not be lower or  $b$  can not be higher if the property of being splitting pair still remains. This assumption regarding split seems reasonable since otherwise another split would exist which overlap would be smaller and the minimal number of entries in the group would not be smaller, i.e., there would be a better split in terms of optimization target of this algorithm.

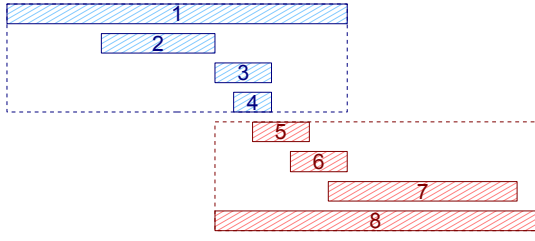


Fig. 3. A split that can be produced by the splitting pair

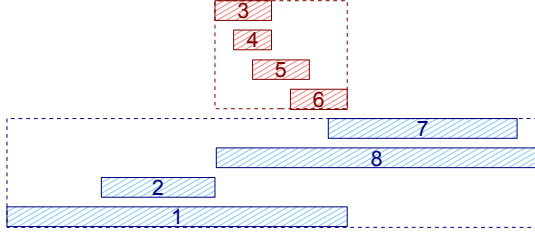


Fig. 4. A split that can not be produced by the splitting pair

## B. Algorithm

The algorithm `EnumerateCornerSplitPairs`(see Algorithm 2) enumerates all corner splitting pairs. The algorithm is based on using two sorted arrays: the first one contains the input entries sorted by the lower bound and the second one contains the input entries sorted by the upper bound. In the main loop of this algorithm, iterations for both arrays are performed simultaneously, so that the property of splitting pair is retained. When a corner splitting pair is found, the `ConsiderSplit`(see Algorithm 3) is invoked. `ConsiderSplit` takes the bounding intervals of groups and maximal numbers of entries which can be placed into groups as its input data. Maximal numbers of entries that can be placed into groups are determined using `EnumerateCornerSplitPairs` by the indexes in the sorted arrays in which the values of splitting pairs are placed. `ConsiderSplit` reveals the split with minimal overlap of group bounding intervals, where the minimal number of entries in group is greater than or equal to  $m$  ( $m$  is minimal number of entries in group). When the split with zero overlap is possible, `ConsiderSplit` chooses the split for which the distance between group bounding intervals is maximal. This property is achieved by allowing the overlap variable to be negative. Let us note that if there are some entries which can be placed into both groups, `ConsiderSplit` considers the split in which the distribution of entries between groups is closest to the uniform one.

The algorithm `DoubleSortSplit`(see Algorithm 1) represents the splitting algorithm in general. At first, it invokes `EnumerateCornerSplitPairs` in order to find allowable corner splitting pair with minimal overlap. Then it distributes entries which can be distributed unambiguously. After that, the rest of entries is sorted by centers of their interval, and they are distributed in a way that makes distribution between groups the most uniform. Since sorting is most time expensive part of this algorithm, it's time complexity is  $O(n \cdot \log(n))$  ( $n$  – number

of input entries).

---

### Algorithm 1 DoubleSortSplit

---

**Input:** Overflowed node

**Output:** Two nodes, at least  $m$  entries in each

- 1: Invoke `EnumerateSplitPairs` in order to find the corner splitting pair with minimal overlap.
  - 2: Distribute entries that can be placed in only one group into groups.
  - 3: Sort the rest of entries by centers of their intervals.
  - 4: Distribute first  $m$  entries to the first group, and distribute other entries to the second group in a way that makes distribution between groups the most uniform.
- 

## C. Application to multidimensional case

The proposed algorithm can also be applied to multidimensional case. Algorithm `MultidimensionalDoubleSortSplit`(see Algorithm 4) represents such an application. At first, it enumerates corner splitting pair along all the axes, and selects the corner splitting pair and the corresponding axis which have the minimal overlap. At second, the entries which can be placed unambiguously are placed. After that the rest of entries are sorted by difference of group area incensement. Finally the split is chosen which has minimal overlap of groups.

## IV. PERFORMANCE TESTS

### A. Experimental setup

All the tests were on run on Core 2 Duo 3 GHz computer with 2 GB of memory with Ubuntu 10.10 32bit. For the implementation of R-tree with various node splitting algorithms GiST[8] framework in PostgreSQL DBMS was selected. GiST generalizes various search trees including R-tree.

### B. Datasets

Each dataset contains  $10^6$  randomly generated intervals. The size of intervals conforms to Gaussian distribution with zero mean and the variance that produces the required level of interval overlapping. The level of interval overlapping varied exponentially from 1 to  $10^4$ . The interval center distribution is determined by the dataset type as follows.

- Uniform dataset. The centers of intervals conform to the uniform distribution along interval  $[0; 1)$ ;
- Gaussian dataset. The centers of intervals conform to the standard Gaussian distribution.
- Uniform cluster dataset. At first, 500 cluster centers, which conform to the uniform distribution along interval  $[0; 1)$ , were generated. After that, for each center 2000 interval centers were generated which offsets from the center conform to the uniform distribution along the interval  $[0; 6 * 10^{-4})$ .
- Gaussian cluster dataset. At first, 500 cluster centers, which conform to the standard Gaussian distribution, were generated. After that, for each center 2000 interval centers were generated which offsets from the center

---

**Algorithm 2** EnumerateCornerSplitPairs

---

**Input:** Set of intervals**Output:** Enumeration of all splits that can be produced with corner splitting pairs by invoking ConsiderSplit

```
1: Sort intervals by lower bound, write the result to array  $a$ 
2: Sort intervals by upper bound, write the result to array  $b$ 
3:  $s1 \leftarrow (a[0].l, b[0].u)$ 
4:  $s2 \leftarrow (a[0].l, b[n-1].u)$ 
5:  $i \leftarrow 0$ 
6:  $j \leftarrow 0$ 
7: {Iterate until finding a first split produced by the corner
   splitting pair.}
8: while  $b[j].u = s1.u$  and  $j < n$  do
9:    $j \leftarrow j + 1$ 
10: end while
11: considerSplit ( $s1, j, s2, n - i$ )
12: while  $i < n$  do
13:    $prev\_s2\_l \leftarrow s2.l$ 
14:    $next\_s1\_u \leftarrow s1.u$ 
15:    $next\_i \leftarrow i$ 
16:   {Find next value of  $s1$  upper bound and the correspond-
     ing value of  $s2$  lower bound which forms the corner
     splitting pair with it.}
17:   while  $next\_i < n$  and  $next\_s2\_l = s2.l$  do
18:      $next\_s1\_u \leftarrow \max\{next\_s1\_u, a[next\_i].u\}$ 
19:      $next\_i \leftarrow next\_i + 1$ 
20:     if  $next\_i \geq n$  then
21:       break
22:     end if
23:      $next\_s2\_l \leftarrow a[next\_i].l$ 
24:   end while
25:   if  $next\_i \geq n$  and  $next\_s1\_u = s1.u$  then
26:     break
27:   end if
28:   {All intermediate values of  $s2$  lower bound form the
     corner splitting pair with the previous value of  $s1$  upper
     bound.}
29:   while  $j < n$  and  $b[j].u \leq next\_s1\_u$  do
30:     if  $b[j].u > s1.u$  and  $b[j].u < next\_s1\_u$  then
31:        $s1.u \leftarrow b[j].u$ 
32:       considerSplit ( $s1, j + 1, s2, n - i$ )
33:     else
34:        $s1.u \leftarrow b[j].u$ 
35:     end if
36:      $j \leftarrow j + 1$ 
37:   end while
38:   {Passage to the next values of  $s1$  upper bound and  $s2$ 
     lower bound.}
39:    $s1.u \leftarrow next\_s1\_u$ 
40:    $s2.l \leftarrow next\_s2\_l$ 
41:   if  $next\_i < n$  then
42:      $i \leftarrow next\_i$ 
43:     considerSplit ( $s1, j, s2, n - i$ )
44:   else
45:     considerSplit ( $s1, j, s2, n - i$ )
46:     break
47:   end if
48: end while
```

---

**Algorithm 3** ConsiderSplit

---

**Input:** Bounding intervals  $s1$  and  $s2$  of two groups, numbers  $n1$  and  $n2$  which represent the maximal numbers of entries that can be placed into each group.**Output:** Updated information regarding the optimal split currently found.

```
1:  $overlap \leftarrow (s1.u - s2.l) / (s2.u - s1.l)$ 
2: if  $n1 \geq m$  and  $n2 \geq m$  and  $overlap < best\_overlap$ 
   then
3:    $best\_overlap1 \leftarrow overlap$ 
4:    $best\_s1 \leftarrow s1$ 
5:    $best\_s2 \leftarrow s2$ 
6:    $best\_n1 \leftarrow n1$ 
7:    $best\_n2 \leftarrow n2$ 
8: end if
```

---

**Algorithm 4** MultidimensionalDoubleSortSplit

---

**Input:** Overflowed node**Output:** Two nodes, at least  $m$  entries in each

- 1: Invoke EnumerateSplitPairs for each axis in order to find allowable corner splitting pair with overall minimal overlap.
- 2: Distribute entries which can be unambiguously placed into only one group in accordance with the corner splitting pair previously found.
- 3: Sort other entries by the difference of group area incensement when adding the entry.
- 4: Distribute the first  $k$  sorted entries to the first group, and other entries – to the second group, so that the minimal overlap between group MBRs over all possible  $k$  is achieved.

---

conform to the Gaussian distribution with zero mean and the variance of  $6 * 10^{-4}$ .

For two-dimensional case the datasets were similar. Rather than scalar random values that were generated in the datasets above, vectors of random values having the same distribution that was used in one-dimensional case were generated. Thus these datasets contained rectangles.

### C. One-dimensional case

The tests have shown that all sorting-based splitting algorithms perform on this datasets almost equally. This is why only one sorting-based algorithm is represented here, namely, the center sorting algorithm. The following node splitting algorithms were included into tests for one-dimensional case.

- Guttman's quadratic algorithm.
- Center sorting algorithm that searches for the split with minimal level of overlap.
- The proposed double sorting-based algorithm.

In order to compare the efficiency of index structures produced by various splitting algorithms, the numbers of node accesses for query execution were measured. 100 small random intervals having size  $10^{-5}$  were generated for testing, and the



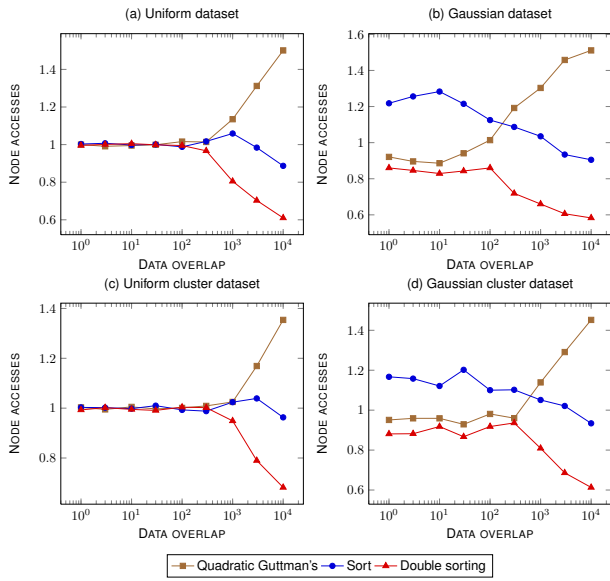


Fig. 5. Comparison of the node access numbers for one-dimensional splitting algorithms

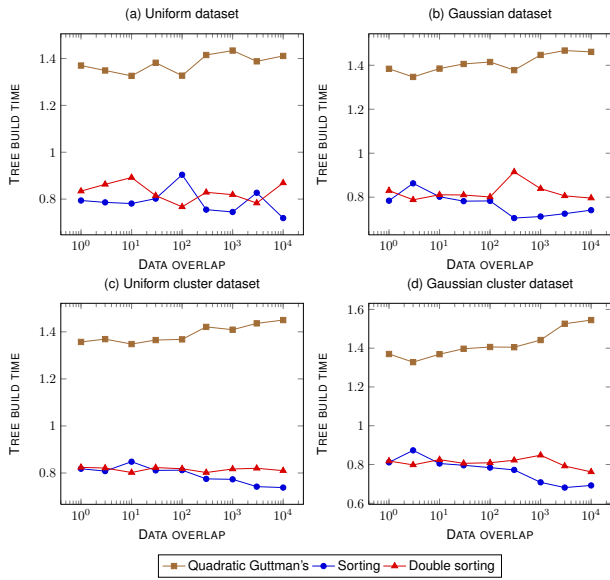


Fig. 6. Comparison of tree building time for one-dimensional splitting algorithms

number of node accesses required for finding intervals in test datasets that overlap with them was measured. In the figure 5, the average number of node accesses is shown. To simplify the comparison, not absolute value of node access numbers is presented, but rather the ratio of that value for particular algorithm to the average value for all algorithms. The measurements were performed for four datasets described in the subsection above, and for various data overlap levels. In the figure 6 the comparison of tree building times is presented. The data is presented in the same manner as for the data access: as a ratio of building time of particular algorithm to

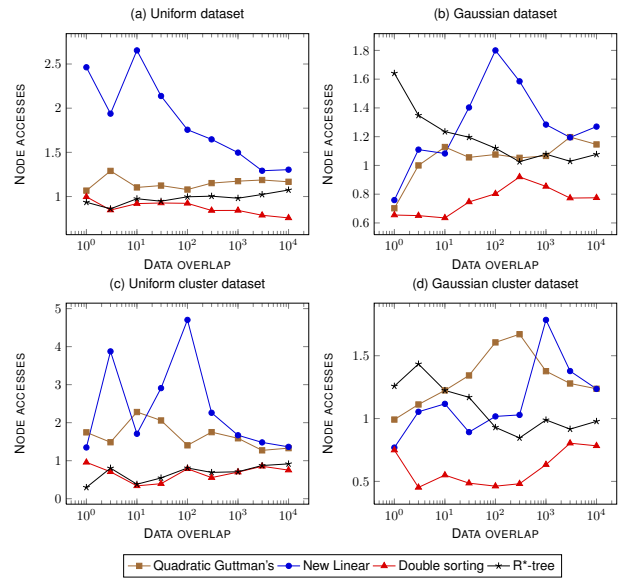


Fig. 7. Comparison of the node access numbers for 2-dimensional splitting algorithms

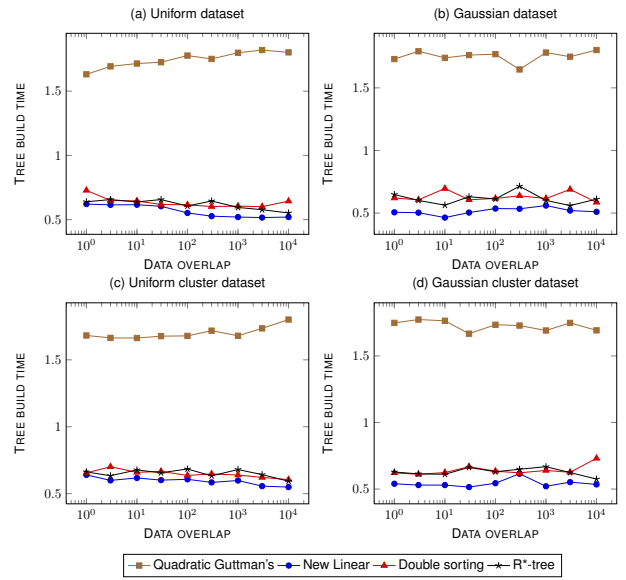


Fig. 8. Comparison of tree building time for 2-dimensional splitting algorithms

the average building time.

We can see that the number of node accesses required for searching in double sorting-based algorithm is almost never greater than this number in other algorithms. With large data overlap, there is significant superiority of double sorting-based algorithm, up to 50%, in comparison with sorting algorithm, and up to 2 times in comparison with Guttman's quadratic algorithm. We can see that tree construction time for double sorting-based splitting algorithm is smaller than that for Guttman's quadratic algorithm, but is slightly greater than the time for the sorting algorithm.



#### D. Two-dimensional case

The following node splitting algorithms were included into tests for two-dimensional case:

- Guttman's quadratic algorithm.
- New linear algorithm.
- Proposed double sorting-based algorithm.
- R\*-tree splitting algorithm.

Numbers of node accesses for query execution and tree building time were compared in a same manner as in the one-dimensional case. In the figures 7 node access numbers are compared. In the figure 8 tree building times are compared. At first, we can see a weaker correlation between relative node access numbers and the data overlapping. And that correlation is decreased with increasing the number of dimensions. We can see that double sorting-based algorithm shows superiority in terms of node access numbers in most test cases. The tree building time of double sorting-based algorithm is close to that of R\*-tree splitting algorithm.

#### V. CONCLUSION

In this paper, new double sorting-based node splitting algorithm for R-tree was proposed. This algorithm was initially developed for better handling of complicated cases in one-dimensional split. The proposed splitting algorithm is based on the notion of corner splitting pair and the algorithm of its enumeration. After that, this splitting algorithm was applied to multidimensional cases.

In one-dimensional case, the tests show superiority of the proposed algorithm in terms of the number of node accesses over Guttman's quadratic and simple sorting-based algorithm. The higher superiority was achieved with larger data overlap due to ability of the proposed algorithm to better handle complicated cases. In two-dimensional case, the tests show superiority in terms of number of node accesses over Guttman's quadratic, new linear and R\*-tree splitting algorithms in most test cases.

#### REFERENCES

- [1] A. F. Al-Badarneh, Q. Yaseen, and I. Hmeidi. A new enhancement to the r-tree node splitting. *J. Information Science*, 36(1):3–18, 2010.
- [2] C.-H. Ang and T. C. Tan. New linear node splitting algorithm for r-trees. In *Proceedings of the 5th International Symposium on Advances in Spatial Databases, SSD '97*, pages 339–349, London, UK, 1997. Springer-Verlag.
- [3] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.
- [4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r\*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331, May 1990.
- [5] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, June 1998.
- [6] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 606–615, Washington, DC, USA, 1989. IEEE Computer Society.
- [7] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.
- [8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95*, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
- [9] K. Kanth, D. Agrawal, A. Singh, and A. E. Abbadi. Indexing non-uniform spatial data. *Database Engineering and Applications Symposium, International*, 0:289, 1997.
- [10] C. P. Kolovson and M. Stonebraker. Segment indexes: Dynamic indexing techniques for multi-dimensional interval data. In J. Clifford and R. King, editors, *SIGMOD Conference*, pages 138–147. ACM Press, 1991.
- [11] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, 31:158–221, June 1999.
- [12] Y. Tao and D. Papadias. Performance analysis of r\*-trees with arbitrary node extents. *Tran. Knowl. Data Eng. (TKDE)*, 16:6–653, 2004.

# Fuzzy matching for partial XML merge

Vladimir Fedotov  
ISP RAS  
Moscow  
Email: vfl@ispras.ru

**Abstract**—In this paper we describe our experience in creating a tool capable to provide traceability of the requirements between different versions of the LSB and POSIX standards. We propose an approach for merging requirement markup without exact matching of the documents by means of arithmetic hashing and context similarity analysis.

## I. INTRODUCTION

In the past several years XML made huge progress, mostly due to its extensions. ISO finally standardized OpenXML and OpenDocument as document formats for office applications, thus making possible long standing dream of simple and robust document exchange in a heterogeneous environment.

At present day there are hundreds of XML extensions [1], applicable to a huge variety of tasks, from relational databases to vector graphics. Among the most important - XHTML providing transition from HTML to XML. Its importance relies on a huge amount of standards, specifications, recommendations and RFCs available in HTML and, with adoption of XHTML, available to processing by means of XML-aware tools.

Our present work is intended to provide requirement traceability between different versions of POSIX and LSB standards. Previous versions of these standards were converted to XHTML with requirements markup for the needs of OLVER [2] and other projects.

Actual markup consists of one or more span tags wrapping requirement-related text nodes. It is important to note here that span can actually divide text node to several chunks, therefore creating a sub-tree in place of one node, thus making impossible direct matching between marked and non-marked documents.

```
<parent>
  unrelated text
  <span requirement_id=1>
    The spice must flow!
  </span>
  unrelated text
</parent>
```

Listing 1. Markup example

So our goal may be defined as follows: given marked up previous version of the document, trace requirements chunks in the next version of the document and, if that trace was successful, merge requirement markup.

## II. PROBLEM DESCRIPTION

Despite being straightforward, this problem actually represents a challenge to existing XML merging tools. Limited to open source Java implementations, we were unable to find a solution robust enough due to various reasons.

First of all, the given task requires zero faults merge, as a fault leads to document corruption. In case of XML merge, a fault is mismatching during the node mapping process, which can easily occur between nodes with the same names or values.

Second, it requires to trace a match between actually different nodes, as the markup can break up a text node in two or more chunks.

Third, the solution should take into account the context of matching nodes. For example, valid requirement for one document isn't valid for another document if it was moved to "Deprecated requirements" section.

Finally, the solution has to be stable enough to work in the production environment. Unfortunately, most of the discovered tools were academic researches discontinued by their authors.

## III. RELATED WORK

Tree diffing and merging remains a very active field of research, presently due to popularity of the bio-informatics. There is a variety of different techniques applicable to very different sets of problems. Very good review of the techniques available for XML was done by K. Komvotzas in [3].

Basically, an XML documents can be merged by direct comparison of their nodes, traversed in the definite order. Discovered conflicts fall into two categories: node insertion and node deletion. Some algorithms also recognize moving and updating, while others simply represent them as a consecutive deletion and insertion. Ruling out these conflicts is actually called a "merge".

Unfortunately, such direct comparison will often provide bogus results because nodes may only differ by their context. There are several approaches developed to compare node contexts.

DiffX algorithm [4] uses fragment matching technique. According to it, a node is considered a match if it has the largest matching neighborhood then any other equal node. Weakness of that approach is in corner cases – it is unable to provide any ruling in case of equal nodes with equal fragments, thus leaving us without guarantees of correct matching. Moreover, there is no simple way to perform a reverse task – to trace a single node by its context, in case we want to match actually different nodes.

A slightly different approach is proposed in [5]. So-called "fingerprinting" is a way of storing node context in a vector of MD5 hashes. A node is considered a match if its fingerprint has the matching neighborhood within a given radius. As opposed to DiffX, it is possible to trace a node by matching its context, but such an approach will produce false results in case of matching several equal chunks with the same context, like brackets or commas enclosing reference links, which often occurs in documents.

Radically different is a three-way merge approach, which makes use of three documents instead of two. Often used in the revision control systems, three-way merge compares two target documents while also considering a "base" document. In practice, it is implemented as a diff between the document  $\alpha$  and the base, and then patching the document  $\beta$  with resulting delta.

In our case it could be implemented by considering the original unmarked document as a base, diffing it with the marked up document, and patching the target document with resulting delta. Diffing the unmarked base with the marked up document will produce the markup anchored to the actual nodes, which can be traced much more effectively in a target document. We are considering implementation of this approach in the future versions of our tool.

#### IV. ARITHMETIC HASHING

To address the issues described above we propose an approach combining two different ways of evaluation of the node equality: by using positional metric and by comparing similarity of their text contents.

Each node has several attributes defining its position in the document: position in the traversal sequence, position among the siblings, depth, index among the identical siblings and overall siblings count. The problem is that these attributes should be properly weighted to be used in the positional metric and approach to provide such weighting isn't clear. Which is more important for node equality: to have same depth or same traversal position? What if the node match candidate have exactly same siblings and depth, but is located in completely different place in document? And if another match candidate is located in exactly same place, but has no matching siblings?

In our implementation arithmetic hash provides us with cumulative positional metric. It uses general approach quite similar to arithmetic coding algorithm [6], but in completely different way.

Starting from the root, an interval is assigned to each node, with a random number from that interval assigned as its label. Each interval is divided by the number of node descendants and each descendant is provided with a range within parent interval according to its position among sibling nodes (fig. 1).

While being extremely simple this hashing technique provides us with positional metric which can be easily evaluated as a distance between the node labels (1). Its deliberately made biased towards matching of the nodes on the same axis, but being extremely sensitive to any changes in the siblings order

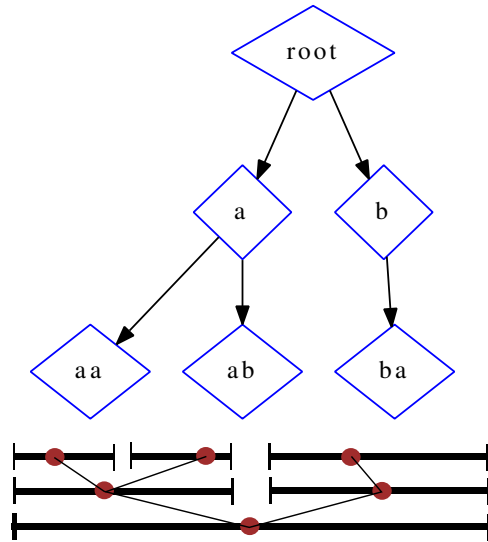


Fig. 1. Labelling example

providing better matching for equal text chunks, as opposed to [5].

$$distance(x, y) = \frac{|label(x) - label(y)|}{\min(interval(x), interval(y))} \quad (1)$$

#### V. CONTEXT SIMILARITY

Bearing in mind that we are dealing with an XHTML documents, it is safe to say that almost any axis in such documents ends with a text node leaf. Therefore any ascendant nodes can be defined by the concatenation of the descendant text node values.

Evaluating similarity between text contexts provides us the similarity metric, that can be used for searching of the proper parent node, which later serves as a context for positional metric. Similarity itself can be evaluated by applying Jaccard index to the tokenized text contents (2).

$$jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (2)$$

Being much faster than commonly used Levenshtein edit distance [7], Jaccard index provides satisfactory results, which can be further improved by taking into account the fact that requirement text should be equal or included into reference text. Therefore in our approach we use slightly modified similarity index (3).

$$similarity(A, B) = \frac{|A \cap B|}{A} \quad (3)$$

#### VI. GLUING IT TOGETHER

While none of the metrics described above can be used effectively separately, being combined they provide surprisingly robust results.

Similarity metric provides good results in case of comparison between nodes with diverse chunks of text, but fails if text contexts are exactly the same, which occurs quite often.

On the contrary, positional metric is ineffective in case of comparison between trees with strong differences, but becomes extremely effective for the small sub-trees.

Wrapping these metrics inside a stable sorting algorithm achieves the desired result. Firstly nodes are being sorted by their positional metric, then by their similarity metric. Therefore similarity metric has the priority over positional one, but in the corner case of several nodes having the same similarity index, they will be ruled out by their label distance, as opposed to [4].

## VII. CONCLUSION

This paper presents a new approach to merge XML documents without exact matching of their nodes by using an algorithm combining node matching, based on the positional metric evaluated as label distance, with text similarity analysis based on the evaluation of modified Jaccard index between reference and target text nodes.

Despite being work in progress, our approach is already showing good results while merging different versions of LSB and POSIX standards as well as RFCs and many others.

In the nearest future we consider implementing three-way merge, based on our current approach and provide an open source Java implementation.

## REFERENCES

- [1] [http://en.wikipedia.org/wiki/List\\_of\\_XML\\_markup\\_languages](http://en.wikipedia.org/wiki/List_of_XML_markup_languages)
- [2] <http://linuxtesting.org/project/olver>
- [3] K. Komvotzas, *XML Diff and Patch Tool*, 2003.
- [4] R. Al-Ekram, A. Adma and O. Baysal, *diffX: An Algorithm to Detect Changes in Multi-Version XML Documents*, 2005.
- [5] S. Ronnau, G. Philipp and U.M. Borghoff, *Efficient Change Control of XML Documents*, 2009.
- [6] J.J. Rissanen, *Generalized Kraft Inequality and Arithmetic Coding*, 1976.
- [7] V.I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, 1966.

# High-level Data Access Based on Query Rewritings

Ekaterina Stepalina

National Research University – Higher School of Economics  
33/5 Kirpichnaya, st., Moscow, Russian Federation  
estepalina@mail.ru

**Abstract**—This paper describes the ODBA problem solution based on query rewriting techniques, introduces the DL-Lite logics for knowledge representation and the query rewriting algorithms for high-level data access. The RQR algorithm's optimization capabilities are considered.

**Keywords**- ODBA; description logic; DL-Lite; query answering, query rewriting, OWL 2 QL

## I. INTRODUCTION

A conceptual interface for accessing the data stored in existing relational databases can be implemented via query rewriting techniques. Built on these techniques, the interface may be independent from DBMS and as well as from particular DB schemes[6]. The development of such interface is an actual problem of raising the abstraction level for working with data and high-level integration of information systems. The ontology representation format OWL 2 QL has been specially designed to use actual database technology for query answering via query rewriting. An efficient query rewriting algorithm RQR[1] was introduced at the international OWLED-2009 workshop: it translates queries to ontologies into the queries to ordinal databases. The data complexity of RQR is no higher than P in the worst case. The additional advantage of the algorithm is that it can be used for more expressive descriptive logics – DL-Lite and higher. This paper describes the ODBA problem, introduces DL-Lite logics and query rewriting techniques, then analyses the RQR optimization capabilities.

## II. ODBA PROBLEM

With conceptual modeling progress (OOP, UML) and system sophistication the need of providing an information system with a high-level interface for working with large amounts of data is appeared. Such interface may be provided if the knowledge domain is represented in an ontology description form (knowledge base). The data access problem though a high-level conceptual interface is called ontology-based data access (ODBA) [1]. The solution must satisfy the following requirements: 1) efficient query processing, which must be ideally executed with the same speed as the SQL queries over existing RDB, and 2) the query processing must use all advantages of relational technologies already used to store data.

## III. ONTOLOGY-BASED KNOWLEDGE REPRESENTATION

Knowledge base, KB - is the knowledge domain description saving the relationships' semantics between concepts. KB allows extracting data stored in a database

(ABox), taking into account the constraints expressed at a higher conceptual level (TBox)[4]:

$$KB = TBox + ABox, \text{ or } K = (T, A) \quad (1)$$

Where TBox (T) – terminological box – the conceptual data model, for instance, Entity-Relationship;

ABox (A) – assertional box – data set stored in a database.

An ontology can be called a particular instance of KB, represented on a formal KB description language. Description logic (DL) of a special expressivity power can be used as a knowledge representation language. The expressivity power is defined by the set of axioms allowed in TBox and ABox. On the one side, the language should be as more expressive as possible to completely describe the knowledge domain. On the other side, the reasoning problems over KB must have an acceptable computational complexity.

## IV. SYNTAX AND AXIOMS OF THE DL-LITE FAMILY

The DL-Lite[1] language family is proposed for conceptual modeling in addition to UML and ER. The DL-Lite syntax:

$$R ::= P_k \mid \bar{P}_k, \quad (2)$$

$$B ::= \perp \mid A_k \mid \geq qR, \quad (3)$$

$$C ::= B \mid \neg C \mid C_1 \sqcap C_2, \quad (4)$$

$$\text{inv}(R) = \begin{cases} P_k^-, & \text{if } R = P_k \\ P_k, & \text{if } R = P_k^- \end{cases} \quad (5)$$

TBox is a finite set of  $C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$  - concept and role inclusion axioms.

ABox is a finite set of  $A_k(a_i), \neg A_k(a_i), P_k(a_i, a_j)$  and  $\neg P_k(a_i, a_j)$  - assertions.

Where  $a_i$ - object name, A – concept name, P – role name, q – integer number.

Interpretation  $\mathcal{J}$  (essentially, the particular instance of KB) is a pair if non-empty domain and an interpretation function

$$(\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}}) : a_i^{\mathcal{J}} \in \Delta^{\mathcal{J}}, A_k^{\mathcal{J}} \subseteq \Delta^{\mathcal{J}} \text{ and } P_k^{\mathcal{J}} \subseteq \Delta^{\mathcal{J}} \times \Delta^{\mathcal{J}} \quad (6)$$

For each interpretation the unique name assumption (UNA) status is also specified. UNA affects on the computational complexity characteristics of  $\mathcal{J}$ :

$$a_i^{\mathcal{J}} \neq a_j^{\mathcal{J}}, \text{ for all } i \neq j \quad (7, \text{UNA})$$

Languages of different expressive power are produced by restricting the set of allowed axioms. The main axioms:

$$(P_k^-)^J = \{(y, x) \in \Delta^J \times \Delta^J \mid (x, y) \in P_k^J\}, \quad (8)$$

$$\perp^J = \emptyset, \quad (9)$$

$$(\geq qR)^J = \{x \in \Delta^J \mid \aleph\{y \in \Delta^J \mid (x, y) \in R^J\} \geq q\}, \quad (10)$$

$$(\neg C)^J \mid \Delta^J \setminus C^J, \quad (11)$$

$$(C_1 \sqcap C_2)^J = C_1^J \cap C_2^J \quad (12)$$

Where  $\aleph$  means the cardinality of the following set.

Additional axioms reflect various relationships used in conceptual modeling:

$$(\geq qR.C)^J = \{x \in \Delta^J \mid \aleph\{y \in C^J \mid (x, y) \in R^J\} \geq q\} \quad (13)$$

$$J \models \text{Dis}(R_1, R_2) \text{ iff } R_1^J \cap R_2^J = \emptyset, \quad (14)$$

$$J \models \text{Asym}(P_k) \text{ iff } P_k^J \cap (P_k^-)^J = \emptyset, \quad (15)$$

$$J \models \text{Sym}(P_k) \text{ iff } P_k^J = (P_k^-)^J, \quad (16)$$

$$J \models \text{Irr}(P_k) \text{ iff } (x, x) \notin P_k^J \text{ for all } x \in \Delta^J, \quad (17)$$

$$J \models \text{Ref}(P_k) \text{ iff } (x, x) \in P_k^J \text{ for all } x \in \Delta^J, \quad (18)$$

Where  $\models$  is a satisfaction relation in KB.

The common denominators of DL-Lite logics are the following – 1) it is not possible to assign particular roles only to certain concepts, that means all roles can be applied to every concept ( $\exists R.C, C \sqsupseteq I$ ); 2) TBox axioms are only concept inclusions and cannot represent any kind of disjunctive information, for instance, that several concepts cover the whole domain.

## V. MAIN PROBLEMS OF WORKING WITH KNOWLEDGE BASES

Given a KB  $K = (T, A)$  one may consider the following fundamental reasoning problems [5]:

### A. Satisfiability

Check whether a model of  $K$  exists.

### B. Instance checking

Given an object  $a$  and a concept  $C$ , check whether  $K \models C(a)$ , or, in other words, whether  $a^J \in C^J$  for each  $J$  of  $K$ .

### C. Query answering

Given a query  $q(\vec{x})$  and a tuple  $\vec{a}$  of objects from  $A$ , check whether  $K \models q(\vec{a})$ , or, in other words, whether  $\vec{a}$  is an answer to the  $q(\vec{x})$  query w.r.t.  $K$ .

The computational complexity of these problems depends on a number of variable and fixed input parameters. The input parameters are: the TBox size,  $|T|$ , the ABox size,  $|A|$ , the  $K = (T, A)$  size, the query  $q(\vec{x})$  size – the number of query parameters,  $|\vec{x}| = N$ .

The combined and data (by the amount of data to be processed) complexity are separately considered w.r.t. reasoning problems. The data complexity is the most important in ODBA problem context, so the TBox size is considered fixed, and the query size is negligible w.r.t. the size of ABox.

## VI. EFFICIENT QUERY ANSWERING IN DL-LITE KBS

The maximal expressive language for conceptual modeling, for which the query answering complexity (data) will not exceed  $P$ , is  $DL - \text{Lite}_{\text{horn}}^{\text{HF}}$  [1]. If UNA is accepted, then query answering in  $DL - \text{Lite}_{\text{horn}}^{\text{HF}}(\text{HN})$  will have the least computational complexity by the amount of data -  $AC^0$ . This feature causes a very important fact:

Given a knowledge base  $K = (T, A)$  satisfying  $DL - \text{Lite}_{\text{horn}}^{\text{HF}}$  with UNA and a conjunctive (with no disjunctions) query  $q(\vec{x})$ . Then  $q(\vec{x})$  and TBox can be rewritten into a union of conjunctive SQL( $q(\vec{x})$ ) queries over ABox only, and the answer for this new query will be sound and complete[3].

Based on this fact, query rewriting allows one to obtain a knowledge base over a traditional database, as well as to work with data at the conceptual level independently from a certain database scheme, and effectively use all advantages provided by modern relational DBMS.

## VII. QUERY REWRITING ALGORITHMS FOR OWL 2 QL AND HIGHER

For information systems working with large amounts of data, mostly performing the query answering problems, the W3C consortium's proposed the OWL 2 QL standard. This standard based on less expressive, than  $DL - \text{Lite}_{\text{horn}}^{\text{HF}}$ , the  $DL - \text{Lite}_{\text{core}}^{\text{H}}$  subset of axioms (another designation -  $DL - \text{Lite}_R$ ). The complexity of all reasoning problems over  $DL - \text{Lite}_{\text{core}}^{\text{H}}$  ontologies does not exceed polynomial. This significant restriction's been added because the equality or inequality of objects in OWL is to be specified explicitly with no UNA (or not UNA) implicit assumption. To keep the reasoning problems' complexity constant and UNA-independent for ontologies built in compliance with the OWL 2 QL standard, it's been decided not to include axioms, which allow one to define function dependencies and numeral restrictions over concepts. These axioms strongly affect the reasoning complexity, which depends on the fact whether UNA or not UNA is assumed in the ontology.

Query rewriting techniques and algorithms are intensively developed for OWL 2 QL to provide mechanisms for high-level conceptual query answering over existing databases.

Currently two algorithms have been designed and implemented[2]: CGLLR and RQR.

The CGLLR algorithm for DL-Lite has been implemented in several systems, such as QuOnto, Owlgres, ROWLKit. The RQR algorithm for DL-Lite+ was introduced in 2009 and implemented in REQUIEM. Both algorithms, CGLLR and RQR, retrieve the same results of query rewriting. During the rewriting process each algorithm produces a large number – about several thousand - UCQ (unique conjunctive query). This results in complicated SQL queries with too many unions, which can be impracticable to DBMS.

The algorithms have been tested on computers with equal configuration. The testing data included 9 ontologies of the  $DL - \text{Lite}_R$  [2] expressivity level, corresponding to the OWL 2

QL profile and used in real applications, such as VICODI project, LUBM, SANAP and other.

An active optimization work on these algorithms is conducted in the following directions:

- Simplifying the initial query  $q(\vec{x})$  through query subsumption check;
- Excluding UCQ, which have no corresponding OWL-RBD mappings.

The experiments[2] showed that in some cases RQR with subsumption checking generates less UCQ, than CGLLR. Moreover, unlike CGLLR, the RQR algorithm can be used for more expressive description logic languages, than DL-Lite.

In whole, RQR works more effectively than CGLLR, supports large amounts of data, complex queries and qualified existential restrictions ( $\exists$ ). With subsumption checking applied to initial queries both RQR and CGLLR generate an equal number of UCQ. However, the subsumption check itself takes time and practically equalizes the result efficiency of RQR and CGLLR in the worst case.

### VIII. FUTURE WORK

The experiment results demonstrate that RQR is more preferable for query rewriting, than CGLLR[2].

For researching into practical usage aspects of these algorithms, first of all, one should find out how much query answering based on described query rewriting techniques is efficient on real databases. The obvious obstacle for query rewriting approach is the need of mapping a conceptual model to a particular database for each database and for each unique model. However, it is an additional abstraction layer

requirement, which is inevitable to raise the abstraction level of data access interface.

In further experiments the testing data must include queries, which are to be transformed into SQL queries to real databases based on prerequisite mappings. One may suppose that the query rewriting algorithm efficiency may also significantly depend on a particular mapping representation. Currently, there are no standards and examined formalisms to define such mappings.

Further optimizations can be applied to RQR: forward and backward subsumption check, query condensation and other. Additional experiments with these optimizations are needed. Besides, full features of OWL 2 QL (especially, data types) must be supported in RQR, and a new series of experiments will be required to get reliable results of checking the RQR efficiency with the complete support for DL – Lite<sub>R</sub>.

- [1] Artale, A.; Calvanese, D.; Kontchakov, R. and Zakharyashev, M. (2009) The DL-Lite family and relations. *Journal of Artificial Intelligence Research* 36 (1), pp. 1-69. ISSN 1076-9757.
- [2] H.P'erez-Urbina, I.Horrocks, and B.Motik. Efficient Query Answering for OWL 2. In *Proceedings of the 8<sup>th</sup> International Semantic Web Conference (ISWC2009)*, Chantilly, Virginia, USA, 2009.
- [3] H.P'erez-Urbina, B.Motik, and I.Horrocks. Tractable Query Answering and Rewriting under Description Logic Constraints. *Journal of Applied Logic*, 2009.
- [4] F. Baader. *Logic-Based Knowledge Representation*. In M.J. Wooldridge and M. Veloso, editors, *Artificial Intelligence Today, Recent Trends and Developments*, number 1600 in *Lecture Notes in Computer Science*, pages 13–41. Springer Verlag, 1999.
- [5] *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2002. ISBN 0521781760. Edited by F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. F. Patel-Schneider.
- [6] Semantic Future by SWUG. [Online]:<http://semanticfuture.net>.

# APPLICATION OF THE FUNCTIONAL PROGRAMMING TOOLS IN THE TASKS OF LANGUAGE AND INTERLANGUAGE STRUCTURES REPRESENTATION

Peter Ermakov  
Institute for Informatics Problems,  
The Russian Academy of Sciences  
IPI RAN  
Moscow, Russia  
petcazay@gmail.com

Olga Kozhunova  
Institute for Informatics Problems,  
The Russian Academy of Sciences  
IPI RAN  
Moscow, Russia  
kozhunovka@mail.ru

*Abstract. The paper considers issues of formal methods in the tasks of knowledge representation including optimization of formal grammars by means of functional programming languages. One of the possible applications of the formal notation given is illustrated by the task of parallel natural texts analysis and comparison.*

*Keywords: functional programming tools, language structures representation, parallel texts analysis and comparison*

## I. Introduction

At present the problem of machine text analysis in natural language is one of the key challenges in the field of information technologies and research. For its solution one involves various interdisciplinary approaches and methods. This caused by the complex character of the research led in the direction: several disciplines are engaged, namely, computer linguistics, artificial intelligence, and formal mathematical methods.

Among linguistic resources that have been developed as a result of such research in the fields of natural language analysis and knowledge acquisition are well-known electronic dictionaries, syntactic parsers, language ontologies, generators of syntactic trees: WordNet, EuroWordNet, Ontolingua, Russian dictionary RusLan, а также Penn Treebank, Ragel, Syntax Definition Formalism, Spirit Parser Framework, SYNTAX, Yacc, etc. This list is quite incomplete, though, it only demonstrates variety of approaches and instruments involved in the procedures of natural language mechanisms analysis and modelling.

Thus, automatization of a set of processes within the language (grammar analysis, syntactic and semantic representations, etc.) and of its interaction with human activities (speech recognition, machine translation, parallel texts comparative analysis, and so on) is one of the up-to-date relevant tasks for several disciplines and domains simultaneously. [i, ii].

But the level of sophistication of the existing approaches to language structures and processes representation scales up with the growing demands to the language models. This induces a search of new approaches to language structures representation and hybridization of the well-functioning old methods.

As an example one can study the difficult and long way of universal grammar search to use it in natural language representations which is dramatic for the optimization of machine translation systems, syntactic and semantic text analysis, texts comparative analysis and preset concepts and relations acquisition, etc. One of the principal founders of the approach to modelling of natural language units, relations, and mechanisms of their interaction by means of formal grammars was Noam Chomsky [iii]. Time passed, and many followers of his approach appeared. They started the above mentioned long search. It led them to the gradual enumeration and revision of the existing grammar types, then — to their complication and adaptation to possible applications, and finally — to dissatisfaction from the formal representation they got as a result, and address to statistical methods that prevailed in this field up to the moment. But soon many experts in their independent research found out that statistical methods don't solve the problem of completeness and accuracy of the natural language structures representation. For this reason today more and more specialists address to hybrid methods which are untrivial in construction but more precise accordingly to the applications needed. Besides, validation of their implementation is to be carried out through all the stages of the language representations, its structures analysis, comparison with the existing patterns, and so on [iv].

Thus, as it was mentioned above, formal grammars is a mathematical apparatus meant for the text analysis. The most interest form the perspective of the problem considered attract regular and context-free grammars [iii]. For each of them a relevant analytical engine exists thanks to which parsing might be conducted automatically, i.e. by means of computational procedures. Each analytical engine possesses a set of advantages and disadvantages. For instance, some of the disadvantages are complex structure, infeasibility of the universal engine design and need in its constant rebuild according to a specific application.



This involves many questions, particularly: is it possible to build a universal mechanism of the natural language structures representation using the existing formal methods and techniques? Is there any probability of minimizing work of experts in the procedures of formal representations and natural language structures comparison and their verification?

In an attempt to answer these and other questions this paper suggests an approach which allows representing natural language grammars (as well as formal ones) in a form not demanding an analytical engine design for text parsing and analysis. Rejection of analytical engines use will give us an opportunity to get rid of a set of technical problems associated with their complex implementation. For instance, it is suggested to enhance formal grammars with the functional programming languages and their tools. One of the possible applications of the suggested formal representation is illustrated by an example of the comparison of natural language parallel texts.

## II. Formal Grammars and Analytical Engines

Consider a selected at random formal grammar  $G = (V_T, V_N, S, P)$ , where  $V_T$  is a finite set of terminals,  $V_N$  is a finite set of nonterminals, start symbol  $S \in V_N$  and  $P$  is a finite set of productions of a form  $\alpha \rightarrow \beta : \alpha, \beta \in (V_T \cup V_N) \wedge \alpha \neq \emptyset \wedge \exists v \in V_N : v \in \alpha$ .

According to Chomsky classification, formal grammars are divided into four types [iii]:

- unrestricted (type 0)
- context-sensitive (type 1)
- context-free (type 2)
- regular (type 3)

The first two types (type 0 and type 1) have no application due to their complexity, except for the context-sensitive grammars which might be used when analyzing natural languages texts excluding the task of compilers building.

The types 2 and 3 on the contrary have plenty of various applications. For example, context-free grammars are used when describing computer languages syntax. Regular grammars are applied for description of the unary constructions: identifiers, strings, constants, assembler languages, command processors, etc.

An analytical engine for regular grammars is finite state automaton. Equivalence of the regular grammar and finite state automaton is proved in the Kleene theorem [v], which allows assuming the concepts of regular grammar, regular expression, and finite state automaton equivalent.

The field of application of regular grammars in the tasks of natural language structures recognition is rather limited. This is related to the uneasiness of finding a regular expression for describing any of the formal languages, not to mention natural language and its structures. We'd like to note though, that regular expressions is a very convenient tool for analyzing short and highly formalized language constructions.

At present basic instruments of formal and natural languages analysis are context-free grammars. Analytical engine for context-free grammars is a one-sided nondeterministic automaton with stack outer memory. In the most trivial case of such automaton's algorithm implementation, it is characterized by an exponential complexity. But iterative upgrade of the algorithm may lead us to its polynomial time value depending on the length of the input set of symbols and necessary for its analysis [vi].

Among the existing context-free languages one can distinguish a class of deterministic context-free languages, which are interpreted by deterministic automaton with stack outer memory. The principal feature of these languages is their unambiguity: it is proved that one can always build an unambiguous grammar for any deterministic context-free language [vi, vii]. Since the languages are unambiguous, they are most useful when it comes to building compilers [vi].

Moreover, among all the deterministic context-free languages exist such classes of languages which allow building a linear recognizer for them. This is the recognizer which time value related to the time for decision-making about a set of symbols belonging to a language has a linear dependency on the chain length [vi]. Syntactic constructions in the majority of the existing programming languages might be classified as ones of the class mentioned. This aspect is very important when developing up-to-date high-speed compilers.

As a rule, the more complex from the mathematical perspective an analytical engine is, the more challenging is its technical implementation. Case with finite state automata (both with memory and without it) isn't an exception. It's well-known that having formal grammar one can build a automaton accepting it [vii, viii]. But this automaton possesses a set of disadvantages when it comes to its application to natural language grammars. For instance, rules of natural language include a plenty of features of natural language structures which are attributive by their nature. Processing of such transformation rules implemented using finite state automaton may lead to the increase of the automaton states and to growth of amount of the transformation rules when changing the states. Moreover, when implementing the above mentioned transformation rules one needs to create a problem-oriented analytical engine to handle processing of the input set of symbols of the natural language structures and to apply interpreting rules to them.

### III. Computational and Functional Grammars

In order to minimize above mentioned challenges the authors suggest an approach which allows representing of natural language grammar rules as functions in mathematical

sense, that is  $A \xrightarrow{f} B$  or  $y = f(x): x \in A, y \in B$ . For making it more convenient and clear to handle natural language grammar rules as mathematical functions (i.e. as a

transformation which is written as  $A \xrightarrow{f} B$ ) we suggest to use such tools as n-tuples to keep necessary attributive characteristics (for example, gender, singular\plural, etc.). Representation of the rules as functions also gives an opportunity to use instruments of the functional programming to build systems of grammar parsing and analysis and interlingual transfer not losing efforts to build such an analytical engine as a finite state automaton.

Consider the approach in more detail.

Representation of grammar rules, or transformation rules, as mathematical functions has the following advantages in comparison with formal grammar apparatus:

- Usage of functional programming tools to build systems of transfer immediately;
- Possibility of higher-order function applications.

It's worth noting that by concept "system of transfer" in the paper we understand software implementation of an analytic engine for processing of transformations expressed as functions.

We'll illustrate it by an example: consider a simple sentence in English «I can swim» and its translation into Russian «Я умею плавать».

Examine transformations expressed as mathematical functions. For the example given we use syntax of the functional programming language Erlang:

```
(1)
trans(i) → я,
trans(can) → мочь,
trans(swim) → плыть;
```

On applying the functional programming tools, one can split an input sentence into separate words (this mechanism isn't given in detail in the paper). Having applied the above given rules immediately, the following result might be obtained:

```
(2) я мочь плыть.
```

Such "translation" is quite unfit to the translation taking into account all the links between natural language structures. However, we'd emphasize that to build such a system of transfer no additional tools were involved except for the system of rules. The whole mechanism of transfer was provided with the functional programming language.

Application of higher-order functions gives an opportunity to pass a function as a parameter to other functions. This allows, for instance, handling the normalization (i.e. putting into normal form – for example, infinitive verb form, noun in singular, etc.) of any natural language structure before transfer.

Using n-tuples as a form of representation of natural language structures enables to generalize attributive characteristics of words and use pattern alternations of such structures in prospect (see an example below). Consider an example of a function which in a case with its argument a noun in singular leaves it without any modifications, but in a case with it in plural, adds an "s" inflexion to the end of the word:

```
(3)
func({X, noun, singular}) → X,
func({X, noun, plural}) → X ++ «s»;
```

Such function might be of use, say, when generating text in English.

That's why we'd like to consider such representation form of natural language information as n-tuples in detail. As one can see from the example above, this approach gives an opportunity of arrangement of attributive features and its further use in transformations. Apart from this, using n-tuples for storage of attributive features of language structures enables us to extract functions according to language structures of various levels of abstraction (for instance, a word, a phrase, a sentence, etc). any of such language structures has its own set of attributes, hence there should be functions which have words, phrases, and so on, as their arguments.

At first sight the above given approach to representation of natural language grammar rules apparently generates a huge amount of transformations even for a narrow domain. It is the case. But one should note that since system of transfer built using functional programming tools cannot be considered a finite state automaton, and, thus, doesn't have any states and rules of transformations between the states, so the amount of transformations isn't dramatic. Absolute clearness (absence of the inner state) of the analytical engine gives an opportunity to perform analysis and synthesis of transformations by an expert in the field of computer linguistics in a convenient mode.

The second distinctive feature of the suggested approach is quite technical one. It is essential to design and implement a relevant analytic engine for every information system which tasks correlate with natural language structures analysis based on mathematical apparatus of formal grammars. As it was said above, this task is rather complex and laborious. However, in the case of functional representation of grammar rules as an analytical machine the environment of the functional programming might be used (for instance, Erlang, Haskell) [ix].

But here's a more formal description of the suggested approach.

Firstly, introduce a concept “Computational and Functional Grammar”. Consider a formal grammar  $G = (V_T, V_N, S, P)$ , where  $P$  is  $\alpha \rightarrow \beta : \alpha, \beta \in (V_T \cup V_N) \wedge \alpha \neq \emptyset \wedge \exists v \in V_N : v \in \alpha$ .

Computational and Functional Grammar (CFG) is a form of notation of the above mentioned formal grammar  $G_{FC} = (T, f, A)$ , where  $T$  is a finite set of parametric n-tuples,  $f$  – transformation function  $f : T \rightarrow T$ ,  $A$  – finite set of atoms.

By parametric n-tuple we mean an n-tuple elements of which might be elements of the sets  $V_N$ ,  $V_T$ ,  $A$  and special symbol «\_».

By atom we assume any unambiguously determined identifier such that  $A \cap (V_N \cup V_T) = \emptyset$ . Atoms are meant for setting attributive characteristics of natural language words (gender, singular/plural, case, etc.). They are specific instrument for simplifying of natural language structures analysis in particular.

We'd like to emphasize that atoms and atomic structures are included into CFG description. Thus, all possible attributive characteristics of language structures are defined in the Grammar. That is, CFG is a formalism sufficient for natural language structures analysis, and there is no necessity for using any of other mathematical tools in addition to it.

Function  $f$  in Computational and Functional Grammars is defined in the table form similar with the Backus-Naur form which is used for setting the rules in context-free grammars.

The symbol «\_» is suggested to denote an n-tuple element which value one may ignore when defining the transformation function. Thus, illustrate the sense of the special symbol «\_» by an example.

Consider a set of atoms  $A = \{noun, verb, plural, singular, ok, not ok\}$  and a function defined as follows:

$$(4) \quad \begin{aligned} f(\{noun, \_ \}) &\rightarrow \{ok\}, \\ f(\{verb, \_ \}) &\rightarrow \{not ok\}; \end{aligned}$$

Then  $f(\{noun, plural\}) = \{ok\}$  and  $f(\{noun, singular\}) = \{ok\}$ , i.e. the function's value will be  $\{ok\}$  regardless of noun is in singular or in plural.

To illustrate the difference in grammar rules representations by example, we'll consider a formal grammar  $G = (\{a, the, dog, cat, chased\}, \{<S>, <NP>, <VP>, <N>, <V>, <DET>\}, <S>, P)$  where  $P$ :

$$(5) \quad \begin{aligned} <S> &::= <NP> <VP>, \\ <NP> &::= <DET> <N>, \\ <VP> &::= <V> <NP>, \\ <DET> &::= a \mid the, \\ <N> &::= dog \mid cat, \\ <V> &::= chased \end{aligned}$$

where  $<S>, <NP>, <VP>, <DET> \in V_N$  and  $a, the, dog, cat, chased \in V_T$ .

In case of functional representation the above defined grammar will be expressed as follows:

$$G_{FC} = (\{<S>, <NP>, <VP>, <N>, <V>, <DET>, a, the, dog, cat, chased\}, f, \emptyset)$$

where  $f$ :

$$(6) \quad \begin{aligned} f(<DET>) &\rightarrow a, \\ f(<DET>) &\rightarrow the, \\ f(<N>) &\rightarrow dog, \\ f(<N>) &\rightarrow cat, \\ f(<V>) &\rightarrow chased, \\ f(<NP>) &\rightarrow f(<DET>) ++ f(<N>), \\ f(<VP>) &\rightarrow f(<V>) ++ f(<NP>), \\ f(<S>) &\rightarrow f(<NP>) ++ f(<VP>); \end{aligned}$$

Symbol «++» denotes an operation of concatenation.

One should pay attention that in the considered example the set of atoms  $A$  is empty. This points to the fact that such grammar doesn't take into account attributive characteristics of words and language constructions. One may also notice that the set  $T$  in the example above is just a joint alphabet of terminals and nonterminals of the initial grammar  $G$ .

#### IV. Task of parallel texts analysis and comparison

At the contemporary stage of design and development of natural language processing systems the main emphasis is merged towards creation of parallel texts (i.e. texts in several languages equivalent by their contents and representation forms) analysis techniques. It generates a set of tasks concerned with their adequate interpretation and application, above all, those are tasks of machine translation and knowledge processing [x, xi, xii].

For that reason we demonstrate capabilities of the functional programming tools when applied in the task of parallel text analysis and comparison, including the task of interlanguage structures transfer from one language into the other [xiii]. As an example we consider texts of the patent claims (in chemical technologies) in German and English respectively:

(7)

- **Claim in German:** *Verfahren zur Epoxidierung einer organischen Verbindung mit wenigstens einer C-C-Doppelbindung mit Wasserstoffperoxid in Gegenwart wenigstens einer katalytisch aktiven Verbindung und wenigstens eines Lösungsmittels, dadurch gekennzeichnet, dass ein Produktgemisch umfassend a-Hydroperoxyalkohole unter Einsatz wenigstens eines Reduktionsmittels reduziert wird.*

- **Claim in English:** *A process for the epoxidation of an organic compound having at least one C-C double bond by means of hydrogen peroxide in the presence of at least one catalytically active compound and at least one solvent, wherein a product mixture comprising [alpha]-hydroperoxyalcohols is reduced using at least one reducing agent.*

When consistently comparing the claims given (for instance, for the sake of confirmation of the patent information and data mining in chemistry) the following transformations were detected:

(8)

(a) *Verfahren zur Epoxidierung* → *A process for the epoxidation*

*N* [verb, nom, neutr, sg] + *Prep* [zu+der, dat, comp, fem, sg] + *N* [dat, fem, sg] → *Art* [indef, sg] + *N* [com, sg] + *Prep* + *Art* [def, 0] + *N* [com,sg]

(b) *ein Produktgemisch* → *a product mixture*

*Art* [indef, masc, nom, sg] + *N* [comp, nom, neutr, sg] → *Art* [indef, sg] + *N* [com, sg] + *N* [com,sg]

(c) *dadurch gekennzeichnet* → *wherein*

*Pron* + *Part* [II f, masc, sg] → *Adv*

The above given transformations are described by means of primitive language markup and a set of grammar attributes. Thus, in case (b) (example (8)) the transfer of German phrase structure «*ein Produktgemisch*» into English one «*a product mixture*» is described as a modification of an article (Art) with attributes «*indef, masc, nom, sg*» (that is indefinite, masculine, nominative case) and a compound noun N from the left part of the transformation into a phrase structure in English with an article and its attributes «*indef, sg*» and two nouns in the right part of the transformation.

However, this technique of transformations notation (in the example above) possesses some disadvantages, namely, awkwardness of the rule itself and need to interpret it with the help of specifically designed analytic engine.

But using functional programming tools (which are the instruments of language structures representation in the case as well) may give one an opportunity to write down the rule (b) from the example (8), as follows:

(9)

(b) *ein Produktgemisch* → *a product mixture*

$v(\{\langle \text{ein} \rangle, \text{art, indef, masc, nom, sg} \}) \rightarrow \langle \text{a} \rangle,$   
 $v(\{\langle \text{Produktgemisch} \rangle, \text{noun, comp, nom, neutr, sg} \}) \rightarrow$   
 «*product mixture*»;  
 fgerman-english(  
 {X1,art,indef,masc,nom,sg},

{X2,noun,comp,nom,neutr,sg}) →  
 $v(\{\text{X1,art,indef,masc,nom,sg}\}) ++$   
 $v(\{\text{X2,noun,comp,nom,neutr,sg}\});$

## V. Conclusion

In the paper a new approach to natural language grammar representation as functions in mathematical sense is considered. Also opportunities of applying functional programming tools to building systems of transfer are demonstrated. Practical application of the approach is viewed from the perspective of parallel texts analysis and comparison (texts from patent and scientific fields).

Further research within the approach and associated tasks may be conducted in the following directions:

- Customizing of the existing representations of the natural language grammars to functional form;
- Creation of problem-oriented system of functional programming to make handling of natural language rules more convenient;
- Enhancement of functional programming tools taking into account needs and tasks of computer linguistics.

- 
- <sup>i</sup> . Козеренко Е.Б. Лингвистическое моделирование для систем машинного перевода и обработки знаний // Информатика и ее применения, №1, том 1. – М.: Торус, 2007. – С.54-65.
- <sup>ii</sup> . Козеренко Е.Б. Глагольно-именные трансформации при англо-русском машинном переводе // Компьютерная лингвистика и интеллектуальные технологии: Труды международной конференции «Диалог 2007» / Под ред. Л.Л. Йомдина, Н.И. Лауфер, А.С. Нариньяни, В.П. Селегея. - М.: Изд-во РГГУ, 2007. – С. 286-294.
- <sup>iii</sup> Chomsky N. Syntactic Structures. — The Hague: Mouton, 1957.
- <sup>iv</sup> . Jacobs, Roderick A. and Peter S. Rosenbaum. English Transformational Grammar. Blaisdell, 1968.
- <sup>v</sup> Клини С.К. Математическая логика. -М.: изд-во Мир, [1967]1973.
- <sup>vi</sup> Дж. Хопкрофт, Р. Мотвани, Дж. Ульман. Введение в теорию автоматов, языков и вычислений = Introduction to Automata Theory, Languages, and Computation. — М.: «Вильямс», 2002. — С. 528.
- <sup>vii</sup> . А. В. Гладкий, А. Я. Диковский, “Теория формальных грамматик” / Итоги науки и техн. Сер. Теор. вероятн. Мат. стат. Теор. кибернет., 10. – М.: ВИНТИ, 1972. – С. 107–142.
- <sup>viii</sup> . Кобринский Н.Е., Трахтенброт Б.А. Введение в теорию конечных автоматов. – М.: Гос. издательство физ.-мат. литературы, 1962. – 405 с.

---

<sup>ix</sup> <http://erlang.org>, <http://haskell.org>

<sup>x</sup> . Козеренко Е.Б. Проблема эквивалентности языковых структур при переводе и семантическом выравнивании параллельных текстов // Компьютерная лингвистика и интеллектуальные технологии: Труды международной конференции «Диалог 2006» / Под ред. Л.Л. Иомдина, Н.И. Лауфер, А.С. Нариньяни, В.П. Селегея. - М.: Изд-во РГГУ, 2006. – С.252-258.

<sup>xi</sup> . Nivre J., Boguslavski I., Iomdin L. Parsing the SynTagRus Treebank of Russian \ Proceedings of the International Conference COLING'2008, Manchester, UK, 2008.

<sup>xii</sup> . Macken L., Lefever E., Hoste V. Linguistically-based sub-sentential alignment for terminology extraction from a bilingual automotive corpus \ Proceedings of the International Conference COLING'2008, Manchester, UK, 2008.

<sup>xiii</sup> .Кожунова О.С. Выявление номинализованных конструкций в параллельных текстах патентных документов на русском и немецком языках // Компьютерная лингвистика и интеллектуальные технологии: Труды международной конференции «Диалог 2009» / Под ред. Л.Л. Иомдина, Н.И. Лауфер, А.С. Нариньяни, В.П. Селегея. - М.: Изд-во РГГУ, 2009. – С.185-191.

# Static Verification “Under The Hood”: Implementation Details and Improvements of BLAST

Pavel Shved  
Institute for System Programming, RAS  
shved@ispras.ru

Vadim Mutilin  
Institute for System Programming, RAS  
mutilin@ispras.ru

Mikhail Mandrykin  
Moscow State University  
misha.bear.1990@gmail.com

**Abstract**—BLAST is an open-source static verification tool used in checking safety properties of C programs. Given a C program with several assertions, which should not fail at runtime, BLAST statically analyzes the program, and either returns a program execution path that leads to violation of one of the assertions, or proves that no assertion is violated. If BLAST fails to prove inreachability of assertions, it may terminate with error, or loop forever. The framework approach employed in BLAST is counterexample guided abstraction refinement (CEGAR) empowered with lazy abstraction.

The first record of BLAST dates from 2002. The tool had been constantly improving until July 2008, mostly by its original creators. Beginning in 2009, we continued working on it as a part of Linux Driver Verification project.

In this article we overview the current status of BLAST: outline the algorithms the CEGAR framework approach is implemented on top of, describe the heuristics used and the technical details of the implementation, and list the external components BLAST relies on. Along with this description, we outline and evaluate the improvements we made since its last release by the original BLAST team, and share our view on the further improvement of the tool.

**Index Terms**—Software verification, safety properties, reachability verification, static analysis.

## I. INTRODUCTION

BLAST is an acronym of “Berkeley Lazy Abstraction Software verification Tool”. It is a C program verification tool that solves reachability problem. Given a C program, a name of the main function (“entry point”) and a name of a label, it reasons if there exists a program execution path that starts at the entry point and reaches the label specified.

It analyzes the program with CounterExample-Guided Abstraction Refinement approach (for details, see [11]). The way it implements CEGAR is known as “lazy abstraction” [12], a novel approach at that time that aims to retain the part of abstraction that should not change, instead of rebuilding abstraction from scratch after each counterexample analysis. This approach gave the name to BLAST, and the article referenced ([12]) presented BLAST in its “Experimental Results” section; moreover, this article does not contain any reference to another source where BLAST was described or mentioned.

The task in subject is computationally impossible, as the halting problem may be reduced to it. The tool therefore does not guarantee its analysis will terminate. BLAST may terminate in runtime if it detects that the program can’t be

analyzed by it, or provide an incorrect result: either “false unsafe” (a program that does not violate the safety property being checked, but is reported as “unsafe”) or “false safe” (an “unsafe” program reported as “safe”). However, if the tool reports an “unsafe”, it also prints the error trace: the path to error location from the entry point, which may be followed through by a human.

To build an open automated system for Linux device drivers described in [13], we needed a verification tool, and that article outlines BLAST as a tool “intended for academic research in software verification”. The experiments demonstrated that its potential may spread beyond mere academic application, serving as a verification engine in an intensive driver checking workflow, as well as provide a ground for research in the domain of static analysis. However, before its potential strength became current a serious work has been done.

A good description of how the algorithms implemented in BLAST work is in [6]. That article contains a step-by-step explanation how a sample program is verified with BLAST, but does not focus on its implementation details.

In this article we describe what BLAST is now, and outline how we improved it since its last official release (version 2.6) by the original team. “Vanilla” algorithms described elsewhere (see [12] and [6]), we list the undocumented but worthwhile improvements the authors of BLAST made in its implementation<sup>1</sup>; we focus, however, on listing our contribution, and evaluate the impact of our improvements in the relevant domains if possible.

### A. Algorithms used in BLAST

The algorithms used in BLAST are briefly described as “lazy abstraction CEGAR, with Cartesian predicate abstraction and LA+EUFG Craig interpolation as predicate discovery procedure”.

To be more specific, this means:

- **CEGAR** — counterexample-guided abstraction refinement, a process of solving reachability problem by constructing a crude abstraction of all possible paths reached

<sup>1</sup>The authors recommend to always configure BLAST to use these improvements instead of “vanilla” algorithms by specifying “-craig 2 -predH 7” in the command line of the tool.

Fig. 1. Assumption as a library call

```
void custom_assume(int condition)
{ if (!condition)
  ENDLESS: goto ENDLESS;
}
```

from the entry point and iteratively refining this abstraction by analysis of plausible paths that lead to the error location, until the abstraction contains no such paths—or a reachable error location is found. For more info, see [11];

- **lazy abstraction** — an implementation of CEGAR characterized by refining only those parts of abstraction that should contribute to proving inreachability after a counterexample analysis. Re-evaluation of other parts of abstraction is avoided as much as possible. For more info, see [12];
- **control-flow graph (CFG)** — lazy abstraction assumes that the program is represented as a finite control-flow graph, i.e. a labeled-transition system. The abstraction is then a “reachability tree”: a prefix tree of all possible paths in the CFG from the entry point, each node (location) being marked with an abstract state. It’s named an “Abstract Reachability Tree”, or simply “ART”.
- **Cartesian predicate abstraction** — representation of the abstract state of a location as a conjunction of zero or more predicates previously discovered. Unlike Boolean abstraction, Cartesian one restricts usual first-order Boolean logic formulæ to conjunction operator only. For more info, see [2];
- **Craig interpolation** — a procedure of building a Craig interpolant: given two Boolean formulæ with an unsatisfiable conjunction, construct an over-approximation of the first one that uses only terms from the second one, keeping the conjunction of the approximation and the second formula unsatisfiable. As of today, BLAST uses this procedure to construct new predicates for its Cartesian abstraction.

## II. IMPLEMENTATION DETAILS

### A. Generic information

Comprising components written in various languages, BLAST has its core part written in OCaml, and it compiles with OCaml 3.11 version. BLAST runs under Linux.

OCaml abstracts away memory operations, automated processing of which took a considerable time, according to profiling results. By tuning some documented options of OCaml runtime, we decreased memory allocation overhead. Sample programs demonstrated a 20% increase in the amount of locations explored.

### B. Program representation

As previously noted, the approach implemented in BLAST requires the whole program to be represented as a finite

control-flow graph. It means that all functions should be inlined, and no recursion is allowed. However, BLAST approaches this in a different way; it uses CIL [16] to build per-function control-flow graphs. During the program analysis, BLAST automatically jumps to the proper function unless it is called via a function pointer. This approach provides more flexibility, and allows to implement heuristics that concern function calls. For instance, BLAST is capable to support recursion with restricted depth. We also implemented a similar bounding even for non-recursive calls, as we had noticed that analysis does not need to traverse deeply in the call graph to succeed in finding bugs. In Linux Driver Verification project, this allowed us to use code generation tools that automatically satisfy unresolved external calls to functions specified throughout the whole kernel without a dramatic analysis quality degradation.

The representation in such form ignores loop structure, and unrolls them into a set of conditional jumps. The `goto` operators are respected as well. This allows programs to undergo serious transformations, and still be checkable by BLAST. We needed a functionality to denote assumptions of the form “from now on, a certain condition holds”<sup>2</sup>. Instead of trying to built this into BLAST as a special directive, we devised a library function that solves this problem; it’s listed on Figure 1. The function is a valid C as well as it does not confuse BLAST with an unnatural endless loop.

As Linux Kernel sources leveraged the whole power of C language and its GNU extensions, we integrated the latest CIL version to BLAST (1.3.7 instead of 1.3.1), and made several minor improvements to it. Now BLAST is capable to read and process drivers of Linux Kernel of version 2.6.37 with just 2% of modules leading to parse errors during the analysis.

### C. Abstract Reachability Tree exploration

As noted above, lazy abstraction approach does not require the abstract reachability tree to present in memory as a separate data structure. Hence, in BLAST the abstraction is stored in a custom data structure as a graph, and the abstract postcondition computation happens at its leaf nodes (also named the “frontier”). If set of possible program states in a leaf is empty, it’s not traversed anymore. When a leaf contains a plausible error location, the counterexample analysis begins, and the reachability tree is then cut so that its only leaf is the one specified by the error path analysis. When a leaf is covered by another leaf which was already processed, the analysis stops in favor of that already happening starting from the covering leaf. It is implemented by storing “reached region” that comprises all the locations reached so far.

The order the leaves are processed in is tunable. Presets include depth-first traversal, breadth-first traversal and bounded depth-first search (traverse in depth-first manner up to depth  $N$ , add the pending nodes to queue and get the next node from the queue). The default method is BFS; possible reason is that

<sup>2</sup>This is useful to specify preconditions for initial data, which may rule out false positives in certain situations.

it allows to find error locations faster, and the experimental data described in [4] prove that BFS allows faster verification than DFS.

Processing a leaf constitutes on determining its region (an over-approximation of all possible program variable states on this path) by incoming edge in the CFG and region in previous location. More on this procedure in Section II-G.

#### D. Counterexample analysis

When a counterexample—a reachable error location, for which the abstraction contains a non-empty set of program states,—is found in the reachability tree, its analysis starts. A sequence of operations that leads to this location from the root node is fetched from the ART. Then, preconditions of all nodes are taken, and static single assignment (SSA) conversion is applied to the resultant formulæ. The formulæ are stored in a custom OCaml data structure.

Interestingly, a path formula is converted to SSA backwards: i.e. the closer the nodes are to the root of the tree the greater the indexes of their variables are. Therefore, different error paths do not have a common prefix in their path formulæ. An optimization opportunity here would be to reverse the indexes and parallelize ART exploration and path formulæ construction. This might also help with alias analysis (see Section II-I).

For the further analysis, the formulæ are converted from custom format to one of formats suitable for external solvers (special modules take care of that). Due to large size of the formulæ the conversion may take a lot of time. It was the case for SMT solvers format. To overcome this, we focused on this conversion, and made it nearly a thousand times faster, which made the conversion overhead negligible compared to the time to perform an actual formulæ analysis. This result also demonstrates that tight integration with solver’s formulæ representation format might not be necessary for a CEGAR-based verification tool.

After proving that the formula is unsatisfiable (hence the counterexample is spurious), predicate discovery procedure starts. These two activities are described in the next two sections.

#### E. Path feasibility checking

External solvers are to decide if the formula is satisfiable (sat) or unsatisfiable (unsat). If error path formula is unsatisfiable, then the counterexample is spurious, and should be analyzed, and the abstraction should be refined; otherwise, there’s an error in the program. For first-order logic in Linear Arithmetic and Uninterpreted Function Symbols theory, formula satisfiability is a computationally hard problem. Thus, careful choice of SAT solver is crucial for building a fast verification tool.

In the BLAST as of 2008 Simplify solver was used; it is a “stack-based” solver<sup>3</sup> (which makes it ideal for analyzing path

<sup>3</sup>Allows to push/pop conjuncts of a formula and analyze the conjunction of formulæ currently on stack; this could assist checking several formulæ that share common parts for satisfiability.

formulæ concurrently with their generation), but it is a legacy closed-source software with serious licensing limitations. After resolving the performance issues in conversion to SMTlib format (see sec II-D), we turned to experimenting with SMTlib solvers, mainly with CVC3, as its LGPL license fits our aim of building an open toolset for software verification.

We noticed that in BLAST it is possible for SAT solver to report “unknown” instead of “unsat”, and these results are indistinguishable for BLAST. Since proving satisfiability of large formulæ is hard, and, at the same time, if a formula is satisfiable, it’s more likely for the satisfying input to be found really quick, the “unknown” result may server as “unsat” if the solver is tuned properly. We tried CVC3, and discovered that by default it runs in a “honest” mode where no unknown results were possible, and it took CVC3 gigabytes of RAM and several minutes to verify a typical formula appearing in driver source code analysis.

It turned out that the main reason for such a low CVC3 performance on many typical BLAST queries was its use of certain quantifier instantiation heuristics. By default, BLAST path formulæ contain quantified axioms used to model memory with aim to rule out some false unsafes when pointer operations are used. Default settings of CVC3 (used for SMT-LIB benchmark) turned complete quantifier instantiation heuristic, which made it try to instantiate every given axiom with every suitable combination of ground terms occurring in the formula. Also one axiom instance may itself contain new ground terms that can be again used for instantiation. Since the typical BLAST formula contains quantified axioms and a lot of terms, the solver spent much time and memory on the instantiations described above.

We used an option to disable complete instantiation heuristic and put a smaller limit on the number of repeated instantiations. This significantly decreased the number of resulting instances and thus time and memory consumption. Then we also disabled some other heuristics regarding quantifiers. It didn’t cause any significant correctness degradation because the axioms rarely helped the solver prove formula unsatisfiability.

As of today, there is no reason to use Simplify anymore, as CVC3, combined with our fixes to integrational components of BLAST, outperforms it.

We also removed predicate normalization from BLAST<sup>4</sup>, as we supposed that solvers should do it much faster. Our experiments confirmed this.

#### F. Predicate discovery

Vanilla algorithm to discover predicates with Craig interpolation looks like this. A path formula is cut into conjuncts (basic blocks are cut apart), and at each cut point the conjunction of all terms before and after formula may undergo Craig interpolation. For LA+EUf theory Craig interpolants always exist ([15]), and an interpolating prover is a tool to find them.

However, instead of running an interpolating prover at each cut point, BLAST first determines “useful blocks”, a subset of

<sup>4</sup>Actually, we added an option to turn it off/on.



operations along the trace that contribute to unsatisfiability of the formula: a minimum set of blocks conjunction of which is unsat, while the rest of the formula is satisfiable. There may be several non-overlapping sets of useful blocks for a trace. Essentially, “useful blocks” are close to unsatisfiability core of a path formula, with two differences:

- **granularity** of predicate selection is operator-wise, i. e. a predicate for one assignment or conditional may only participate as a whole, while only a part of it may belong to unsatisfiability core<sup>5</sup>;
- **regions may participate** in “useful blocks”. A region, by construction, is an over-approximations of the path formula to the location it is assigned to. So, instead of analyzing the trace prior to a certain location, if a conjunction of the region in this location (computed by previous refinement procedures) and the part of the formula past this location is unsat, then we treat the region as if statement, and nominate it as a part of useful block set.

The bits of formula extracted this way may themselves become predicates for the abstraction (as in one of the steps in SLAM tool [1]). However, BLAST goes further, and runs the interpolating prover for each cut point between blocks in each of the “useful block” sets, treating each set as a small error trace. This way it only calls interpolating prover as many times, as there are these useful blocks, and the formulæ for it to handle are much smaller.

To determine these “useful blocks” BLAST joins predicates in path formula one-by-one, beginning from the last, until their conjunction becomes unsat. Then the latest block joined is a useful one. The next useful block is found with the same procedure, but the first useful block found is added to each conjunction. The procedure repeats recursively until the set of blocks found so far and the next useful block alone form an unsatisfiable conjunction.

For stack-based solver, such as Simplify, joining predicates one-by-one is straightforward. For SMT solvers all intermediate conjunctions were to be checked separately. We noticed, however, that conjunction of all blocks from the end of the trace up to  $i$ -th one is a monotonous function of  $i$  (the more blocks you join, the more likely their conjunction is unsat), and binary search may be applied to find the next useful block. We implemented the binary search for SMT solvers, and we also implemented caching for the predicates converted to SMTlib format. For a complex sample cxausb driver the number of calls to SMT solver was decreased from 32630 to 831, thus reducing the overall verification time of this sample by the factor of 7.

Craig interpolants for each cut point in small “traces” constructed from each of the useful block sets are calculated and added to lists of potential predicates in the locations of the real trace between the first and the last useful blocks, and to the locations in their ART subtrees. The negations of the

interpolants are not added at this point, but they will be taken into account during the refinement.

To perform the interpolation, an Apache-licensed CSIsat prover [9] is used, which takes input in “FOCI” format. It sometimes outputs interpolants with real arithmetic (for instance, it may print “ $x < 0.1 \cdot y$ ” instead of “ $10x < y$ ”); in these cases, BLAST ignores its output and finds less predicates hoping that the rest would be enough to prove the safety of a program.

Each predicates is encoded as a Boolean variable, and each region is stored as a BDD (binary decision diagram) over these variables.

### G. Abstraction refinement

After an error path was encountered and analyzed, the analysis in the ART subtree of the node corresponding to the “useful” block closest to the root is restarted, and the nodes in it are removed from the queue (also known as “frontier”).

To calculate the region of a frontier node, the “abstract postcondition” procedure described in [12] is used. For each of the predicates discovered for this location it is tested if the precondition of the operation along the incoming edge, given the predicate is assumed, is implied by the calculated region in the parent node. To check satisfiability of the formulæ built this way the same SMT solver is used as in the trace analysis.

Since all the data required by such a refinement are local, the exploration of the state space may be made concurrently. We have not implemented this for BLAST, however, a research in this direction yields promising results [14].

After predicate for a node is verified, the node should be tested for coverage. For this, it constructs a BDD that denies that the calculated region for this node implies the reached one, and checks it for truthfulness via BDD, each predicate from the Cartesian abstraction being represented as a distinct BDD term. If this crude check fails, a more precise one with use of the SAT solver is performed; it takes into account that predicates share variables, and are not independent from one another. If a node is not covered, its children are added to the frontier, and the reached region for the location is updated.

### H. Configurable verification

In BLAST it is possible to use lattice-based data-flow analysis to aid CEGAR. Lattices are known to over-approximate the feasible program states, so they may be used to rule out infeasible paths in combination with usual CEGAR analysis to analyze the rest. BLAST contains several such lattices, and only one of them (SymbolicStore) is a generic-purpose lattice that fits all C programs; it is capable to store information on concrete values of integers and structure fields as well as perform shape analysis [5].

To utilize capabilities of lattice-based data-flow analysis, BLAST extends the structure of node’s region beyond the usual conjunction of interpolants. The region in BLAST is a tuple of CEGAR’s predicate constraint, and of several lattice elements, the set of lattices being configured by user. If any of tuple elements is  $\perp$  (or false, for predicate regions), then

<sup>5</sup>It is especially important for more complex formulæ: when alias analysis (see Section II-I) or other techniques (such as [17]) are used.

the further path exploration is not necessary, since one of the means has proved it infeasible.

As for coverage checking, the lattice-aided verification contained a severe issue: the  $stop^{join}$  operator was hardcoded; it made BLAST nominate a single joined region as reached instead of a set of regions. This cuts feasible program paths, since SymbolicStore lattice regions are not a powerset domain [7] (while predicate regions are). Also,  $stop^{join}$  made the number of false safes too big for a certain environment model<sup>6</sup>, so we implemented  $stop^{sep}$  which checks coverage against a set of reached regions, and several versions of  $merge$  operator: join at meet-points (merge-join), join at equal predicates (merge-pred-join)<sup>7</sup>, and no join (merge-sep). After experiments we chose  $stop^{sep}$  with merge-pred-join as our default setting.

As a result, the runtime of BLAST with a SymbolicStore lattice had a 50% increase, but the precision was improved significantly: the amount of true unsafes increased by 20% approximately.

This concept of combining different operators in the exploration of state space of a single program in a configurable way was then developed by one of the authors of BLAST in the other tool, CPAchecker [8]. Our experience demonstrates that while it's not trivial to add more operators the BLAST implementation is loosely-coupled, and it is only lack of syntax and framework sugar what prevents a developer from configuring such operators easily, but the changes one is required to make are not dramatic.

### I. Alias analysis

BLAST employs flow-insensitive may-alias analysis for more precise reasoning about pointer assignments. As pointed out in [5], the analysis is “home-brewed”, and we’ll describe the algorithm here briefly.

The alias analysis starts when the first feasible error location is found. Originally, BLAST performed this costly procedure at the beginning of analysis of a program, but for programs with unreachable error locations, or in cases when all error paths are ruled out by lattice analysis (see Section II-H), but we fixed this.

First, the whole program is analyzed, and the aliasing relation is calculated: if  $x$  may point to  $y$  at any point of program, then “may-alias( $x,y$ )” is true. The relation is not reflexive: while  $x$  may be any identifier,  $y$  should refer to a concrete memory location (stack- or statically-allocated memory, or a location with a `malloc()` call). An over-approximation is built by analyzing each assignment (if  $x$  is assigned an address of  $z$  then  $x$  may point to  $z$ <sup>8</sup>), and closing it transitively (i. e. if  $y$  is assigned to  $x$ , and  $y$  may-point to  $z$  then  $x$  may-point to  $z$ ). This way, an over-approximation of an “ideal” may-aliasing relation is built.

<sup>6</sup>Environment models are “main” functions generated based on templates for Linux device drivers. For more see [13].

<sup>7</sup>Predicate equality was tested via BDDs that stored them.

<sup>8</sup>Pointer operators are ignored at this point, only identifier names are essential here.

Fig. 2. Verification of this program requires more lvalues than it contains

```

void mutex_lock(struct mutex* mtx)
{ assert (*mtx == 0);
  *mtx = 1;
}
void mutex_unlock(struct mutex* mtx)
{ assert (*mtx == 1);
  *mtx = 0;
}
int main()
{ struct mutex* m;
  mutex_lock(m);
  mutex_unlock(m);
}

```

Each expression is encoded as a bit-vector, and the relation is stored in the a BDD. If  $x$  aliases  $q$ , and the expressions are encoded as vectors  $X$  and  $Q$  respectively, then true value for bit-vector  $(X, Q)$  is inserted into BDD.

When the path formula is constructed, and an assignment  $*y=q$  appears in it, alias analysis comes into play. It queries each lvalue (a non-constant expression that denotes a concrete value in the writable memory) encountered in the program if  $y$  may-alias its base identifier. For each lvalue  $x$  it may alias, the following expression is added to the formula in addition to the usual predicate:

$$((y = x) \rightarrow (*x = q)) \wedge ((y \neq x) \rightarrow (*x = *x_{old})) \quad (1)$$

where  $*x_{old}$  is the previous instance of expression  $*x$  in the SSA form. The expression means “if  $y$  really points to the same place  $x$  does, then the value of  $*x$  also becomes  $q$ . Otherwise, this assignment does not change the value of  $*x$ ”.

The lvalues iterated should not be constrained by those encountered in the program. For instance, consider the following program on Figure 2. To verify it, we need to consider  $*m$  as an lvalue, while the program does not contain it. BLAST has a functionality to close the set of lvalues of a program under dereference and taking a field (for structures) operations up to a specified number of dereferences. The depth of such a closure is required to be at least one for the program shown above, but it already prohibitively increase the number of iterations over lvalues, beyond the sensible time limits.

Some may-aliases are also must-aliases. For instance, CIL frontend generates additional variables for assignments to complex lvalues that involve multiple dereferences and field takings. These variables are known to alias only one single variable, and the expressions like (1) generated for them will not contain any disjunctions, and unconditionally assign the value to the must-alias.

We tried to decrease such a number of iterations over lvalues by withdrawing must-aliases from the set of lvalues, then by adding all “const” values to the set of must aliases (to withdraw even more), but we could not make the iterations fast enough. This improved the speed of alias analysis alone

by factor of hundreds, but even this wasn't fast enough. Using faster data structures might help, but we think that a qualitative research boost should precede fast verification of pointer-abundant programs.

### J. Interaction with user

UI was aimed to satisfy a user that looks at the console output: plain text printing of debug information, analysis work, statistics and reports mingled together. We did not change this much, but tuned the output of error trace and verdicts to fit automatic processing of it. Now the external tools may read the verdict, and the error trace with additional information. This is especially useful when the exploration is cut due to function call depth limit a user specified, because it would be unclear from the trace that the limit was enforced rather than the function is not found.

### K. Infrastructure

We added regression tests based on situations that occur during Linux device drivers analysis. They contain both expected and current results, for tracking improvements as well as degradations.

External SAT solvers are connected through a special layer, that allows parallel execution of queries to the external tools. For instance, CVC3 is known to have a lag between it outputs an answer and finished the work (perhaps, due to complicated resource deallocation), but the layer between BLAST and a solver does not make BLAST wait and reap the process. Ditto for interpolating prover.

BLAST contains a lot of dead code. Only a narrow set of options is supported: some configurations do not work at all, and terminate with an exception unconditionally. We did not try to eliminate it; one of the reasons is that it contains surprises: for example, we were going to implement the closure under dereferences we described in Section II-I on our own, but we suddenly found the working code for this commented out.

1) *External components summary*: Default shipment of BLAST includes:

- **CUDD** package — utilities for binary decision diagrams. Implemented in C, distributed under MIT-like license.
- **CVC3** solver [3] — proves (in)satisfiability of various formulæ. Implemented in C++, licensed under LGPL. Communicates to BLAST via SMTlib competition format.
- **CSIsat** interpolating prover [9] — computes Craig interpolants for LA+EUUF. Implemented in OCaml, licensed under Apache. Communicates via FOCI-like interface (which is supported, for instance, by MathSAT [10] interpolating prover as well).
- **CIL** C frontend — converts C program into syntax tree stored as OCaml structures. Implemented in OCaml, licensed under Apache.

### L. Known limitations

BLAST does not support assignments of structures as a whole; does not support function calls by pointer (although some dead code on this matter is included); ignores inline assembly; does not provide automatic deduction of properties involving reasoning about lists and other complex pointer-based structures; does not support arrays, treats each array field as a separate identifier, and can not associate  $a[i]$  and  $a[j]$  if  $i = j$  ( $i$  and  $j$  being the variables); can not reason about pointer inequalities; does not have a fast aliasing solution; ignores short logic in conditional statements; does not cope with interpolants with real numbers; lacks automatic modularization, and always analyzes the program as a whole.

## III. EVALUATION

To evaluate our improvements, we compared how the latest BLAST version from the original developers and our version performs on Linux device drivers from `media/` folder of 2.6.37 kernel, and with a simple rule that checks if the mutex locking is correct. Each launch of BLAST was limited with 15 minutes of CPU time and 1 Gb of memory. To make the older BLAST work with our newest tools, we merged several integrational fixes to it, and we had to merge the latest CIL frontend as well, as the default frontend in the older BLAST can process zero drivers. The results of the comparison are in the Table I.

The results demonstrate that the new version of BLAST is capable to find three times more errors<sup>9</sup> (and the newer version found all the five errors found by the older one), and total speed was improved by the factor of 5, given that the precision of BLAST has increased (see Section II-H). The number of drivers that were reported as neither safe nor unsafe is two times less than those of the original version. With newer version, only 30 drivers exceeded resource limits (only two of them timing out), while the older version ran out of allowed resources in 52 cases, and it means that 22 out of 52 the most complex samples were successfully verified under the same constraints.

In the `media` folder the new frontend has eliminated all the parsing errors; however, a more exuberant evaluation of the newer BLAST demonstrates that as much as 1.1% drivers are still not parsed by BLAST in 2.6.37 kernel. Compared to the original BLAST that can process zero Linux kernel drivers without special patches applied to their source code, this is quite an improvement.

## IV. CONCLUSION

Having started from BLAST 2.6 of 2008, we implemented a lot of fixes to BLAST, which improved its productivity on industrial code base (Linux device drivers) by a factor of more than five (as the evaluation in Section III demonstrates), making it, at the same time, more precise and capable to find more errors, as well as more tolerant to the C code it

<sup>9</sup>The correctness rule was intentionally weak, so most of these errors are not kernel bugs, but they are valid if approached as mere assertion violations in C programs.

TABLE I  
EVALUATION OF THE ORIGINAL AND THE CURRENT VERSIONS OF BLAST

BLAST version	Total	Failures	SAFE	UNSAFE	Total time	Timed out	Memory limit	Other failures
Original	389	110	274	5	11.5 hours	36	16	58
Current	389	57	317	15	2.1 hours	2	28	27

parses. The precision improvement is not just ad-hoc, caused by optimized resource consumption: the algorithms themselves were improved as well.

During our experiments, we succeeded in utilizing a generic SMT solver, and demonstrated that formulæ conversion from an internal verification tool’s format to SMTlib competition format for programs as large as Linux device drivers takes negligible time compared to other activities.

We learned also that BLAST is extensible enough to implement more powerful verification algorithms, albeit it is not a straightforward task for a developer. Thus, the weaknesses of BLAST may be overcome, and it’s too early for BLAST to be considered obsolete.

#### REFERENCES

- [1] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4 false alarms. In *Conference on Formal Methods in Computer Aided Design, FMCAD 2010, Lugano, CH*, 2010.
- [2] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. *Proc. TACAS*, page 268–283, 2001.
- [3] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19<sup>th</sup> International Conference on Computer Aided Verification (CAV ’07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
- [4] D. Beyer, A. Cimatti, A. Griggio, M.E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 25–32, nov. 2009.
- [5] D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy shape analysis. *Proc. CAV, LNCS*, 4144:532–546, 2006.
- [6] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, 2007.
- [7] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th international conference on Computer aided verification, CAV’07*, pages 504–518, Berlin, Heidelberg, 2007. Springer-Verlag.
- [8] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. Technical report, School of Computing Science, Simon Fraser University, 2009.
- [9] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUf. In *CAV*, pages 304–308, 2008.
- [10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The maths4smt solver. In *CAV*, pages 299–303, 2008.
- [11] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Proc. CAV, LNCS*, 1855:154–169, 2000.
- [12] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70. ACM Press, 2002.
- [13] Alexey Khoroshilov, Vadim Mutilin, Vladislav Shcherbina, Oleg Strikov, Sergei Vinogradov, and Vladimir Zakharov. How to cook an automated system for Linux driver verification. In *2nd Spring Young Researchers’ Colloquium on Software Engineering*, volume 2 of *SYRCoSE 2008*, pages 11–14, 2008.
- [14] Nuno P. Lopes and Andrey Rybalchenko. Distributed and predictable software model checking. In *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation, VMCAI’11*, pages 340–355, Berlin, Heidelberg, 2011. Springer-Verlag.
- [15] K.L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.
- [16] George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.
- [17] Pavel Shved. On reasoning about finite sets in software model checking. In *4th Spring Young Researchers’ Colloquium on Software Engineering, SYRCoSE 2010*, pages 17–26, 2010.

# Detecting C Program Vulnerabilities<sup>1</sup>

Anton Ermakov, Natalia Kushik

dept. of Information Technologies

Tomsk State University

Tomsk, Russia

antonermak@inbox.ru, kushiknatalya@yahoo.com

**Abstract**—C/C++ language is widely used for developing tools in various applications, in particular, software tools for critical systems are often written in C language. Therefore, the security of such software should be thoroughly tested, i.e., the absence of vulnerabilities has to be confirmed. When detecting C program vulnerabilities static source code analysis can be used. In this paper, we present a short survey of existing software tools for such analysis and show that for some kinds of C code vulnerabilities this analysis is insufficient. Thus, we briefly present an approach for *SPIN* based approach for vulnerability detection which may be useful in some cases.

**Index Terms**—C programming language, software vulnerability, static/dynamic detection method

## I. INTRODUCTION

THE problem of computer-aided software testing becomes important as the complexity of software tools increases and programs written in C/C++ programming language are often used in many critical systems. The security of such software should be thoroughly tested, i.e., the absence of vulnerabilities has to be confirmed. There are two different approaches for vulnerability testing: static and dynamic methods. In this paper, we present a short survey of existing tools based on static vulnerability detection methods and show that for detecting some vulnerabilities, for example a buffer overflow vulnerability, *SPIN* [1] based approach may be more appropriate.

The structure of the paper is as follows. Section II contains preliminaries. Section III is devoted to static code analyzers: a short survey of existing tools for static vulnerability detection is presented in this Section. Section IV discusses a *SPIN* based approach for vulnerability detection while Section V concludes the paper.

## II. PRELIMINARIES

A program *vulnerability* is a property of the program that allows a user to disturb confidentiality, integrity, and/or availability of this software. Given a set of vulnerabilities (features) of a C program, if the program has none of these

features then the program is said to be *safe* w.r.t. the given set of vulnerabilities; otherwise, the program is *unsafe* w.r.t. this set of vulnerabilities. Vulnerability detection methods can be classified as static and dynamic methods [2]. When static detecting methods are applied the source code is analyzed without running the program while dynamic detection methods require the program of interest to be executed.

Given a C program, in this paper, when illustrating the approaches, we consider the following types of possible vulnerabilities: type overflow, type conversion overflow, array overflow (incorrect array index), string overflow which can be considered as different types of a buffer overflow vulnerability and double free vulnerability. All these types of vulnerabilities are specified in details in [3]. Type overflow occurs in a C code when a variable  $v$  is defined as a variable of type  $t$  and the value  $e$  of this variable when executing the code can exceed the maximal value for type  $t$ . It can occur when a given expression  $e$  is assigned to a variable  $v$ , i.e., the C code has an instruction  $v = e$ , and in general, the maximal value for type  $t$  might be different for different platforms and operating systems. An array overflow takes place when a programmer deals with an array  $a$  that has  $size_a$  items while using a variable  $a[i]$  for  $i \geq size_a$ . When analyzing student software tools implementing well-known array algorithms such as different sorts and/or search of minimal/maximal array item, we noticed that many those programs are unsafe w.r.t. type overflow and array overflow (incorrect array index) vulnerabilities. In order to estimate whether existing static methods can detect type overflow and array overflow vulnerabilities we consider three student implementations of array algorithms and run existing tools for detection of such vulnerabilities. In the next section, we present a short survey of existing tools for static code analysis and their outputs for several vulnerable student programs. We then show that some of such vulnerabilities can be detected using *SPIN* based approach.

## III. STATIC CODE ANALYZERS

When estimating the security of student implementations of array algorithms we considered the following tasks: calculating the average value of integer array items, the bubble sort, the insertion sort. C implementations of these programs are specified in the Table 1 which is divided into three

<sup>1</sup> This work is partly supported by RFBR-NSC grant № 10-08-92003

sections. Table 1.1 contains a C implementation of calculating the average value of integer array items (Program 1), Table 1.2 contains a C implementation of the bubble sort (Program 2) while Table 1.3 contains a C implementation of the insertion sort (Program 3). Program 1 has a type overflow vulnerability in the line

```
sred+=a[i];
```

There is no check in Program 1 if *sred* variable value does not exceed the maximal value of the type *unsigned short*; in this paper, the maximal value equals 65536 and each *unsigned short* variable occupies two memory bytes. Programs 2 and 3 have an array overflow vulnerability, since array indexes of arrays *a* and *arr* are not checked whether they exceed the number of array items.

Program 1 – C implementation of calculating the average value of integer array items	<pre>int main(){   unsigned short n=0, a[10];   printf("Enter size of array,   please:");   scanf("%d",&amp;n);   for (int i=0; i&lt;n; i++)   {printf("%d. ",i);   scanf("%d",&amp;a[i]);   }   unsigned short sred=0;   for (int i=0; i&lt;n; i++)   {   sred+=a[i];   }   sred/=n;   printf("Sred:%d",sred);   system("pause"); return sred; }</pre>
--	---

Table 1.1 C implementation of array algorithms (Program 1)

Program 2 – C implementation of the bubble sort	<pre>int main() {   unsigned short j=0,i=0,n, a[10];   cout&lt;&lt;"Enter integer, please:";   cin&gt;&gt;n;   for (i=0; i&lt;n; i++)   {     cout&lt;&lt;i&lt;&lt;" ";     cin&gt;&gt;a[i];   }   unsigned short temp;   bool t = true;   while (t==true)   {     t = false;     for (j=0;j&lt;n-1; j++)     {       if (a[j]&gt;a[j+1])       {         temp=a[j];         a[j]=a[j+1];         a[j+1]=temp;       }     }     t=true;   } }</pre>
---	--

	<pre>   } } for (i=0;i&lt;n; i++) {   cout&lt;&lt;i&lt;&lt;"="&lt;&lt;a[i]&lt;&lt;endl; } system("pause");return 1; }</pre>
--	---

Table 1.2 C implementation of array algorithms (Program 2)

Below we describe the outputs of several static source code analyzers that have been run against C implementations in the Table 1.

A. *ITS4* is a static code analyzer that has been developed in USA by the *Cigital* company in 1992 [4]. The *ITS4* is a tool for static detection vulnerabilities in C/C++ programs. The tool can be executed under Windows or Linux operating systems.

Program 3 – C implementation of the insertion sort	<pre>int main() {   unsigned short length, key,   arr[10];   int i=0, j=0, tmp=0;   cout&lt;&lt;"length:";   cin&gt;&gt;length;   for (i=0; i&lt;length; i++)   {     cout&lt;&lt;i&lt;&lt;" = ";     cin&gt;&gt;arr[i];   }   for (i=0; i &lt; length; i++)   {     tmp = arr[i];     for (j=i-1;j&gt;=0 &amp;&amp; arr[j]&gt;tmp;j--     )       arr[j+1] = arr[j];     arr[j+1] = tmp;   }   for (i=0;i&lt;length; i++)   {     cout&lt;&lt;i&lt;&lt;"="&lt;&lt;arr[i]&lt;&lt;endl;   }   system("pause"); return 1; }</pre>
--	---

Table 1.3 C implementation of array algorithms (Program 3)

When analyzing a given C code the *ITS4* relies on its database of potentially dangerous C functions and if there is a call for such dangerous function in the given code the *ITS4* returns a corresponding report with some recommendations about proposes (preferable changes) in the code. The *ITS4* tool is a free software tool that can be easily downloaded from web-site [4]. We executed *ITS4* against Programs 1, 2, 3 (Tables 1.1, 1.2, 1.3) and the *ITS4* has detected two calls for

dangerous functions. Those are `scanf()` and `printf()`, in particular, the *ITS4* has reported that `scanf()` is a function of a high risk for a buffer overflow vulnerability.

**B. *Flawfinder*** is also a static C/C++ code analyzer that has been developed by David A. Wheeler in May, 2004 [5]. *Flawfinder* “scans” a given code and similar to the *ITS4*, has a list of potentially dangerous instructions of a code. Given a code, selected dangerous instructions (if any) are then ordered according to the risks. The *Flawfinder* report for a programmer points out the calls for dangerous functions and proposes a way for changing the code. However for the above Programs 1, 2, 3 the *Flawfinder* report has only one dangerous function – `system()` and the recommendation “try using a library call that implements the same functionality if available”.

**C. *Graudit*** is a tool that can also help to statically detect several C code vulnerabilities [6]. In order to run this tool it is necessary to call utility *Grep* under *Unix* operating system. As usual, there can be several options how to run this utility but in the simplest case only the path to `cpp` file has to be specified. As a result, a colorful report will appear where for a given C program, some dangerous instructions are blue colored. One can also manually add more instructions into the database of dangerous functions. For each program in Table 1 the *Graudit* colored functions `scanf()`, `printf()` and stream input/output operators `cin` and `cout`.

**D. *CppCheck 1.46*** is a tool with the original name *C++check* that has been developed by *Daniel Marjamäki* and *Cppcheck team* from 2007 until 2010 [7]. The *CppCheck* utility is specialized for memory leakage vulnerabilities. As it is mentioned in [7] *CppCheck* has detected 21 errors in the Linux Core and many other errors in free software. The *Cppcheck* is also a free software tool under the conditions of the *GNU General Public License*. We have run the *Cppcheck* against above Programs 1, 2, 3 and the output message “No errors found” has been returned.

**E. *AEGIS*** is another tool for static detection vulnerabilities in C/C++ programs [8]. The *AEGIS* has been developing in Digitek Labs since 2008. This laboratory is strongly connected with Saint-Petersburg Polytechnic University, Russia. One of the advantages of this tool is that the *AEGIS* supports vulnerability detection for several files simultaneously if they are united in one project. The *AEGIS* detects vulnerabilities that can often occur in C programs, such as memory leakage, incorrect pointers, incorrect array indexes, uninitialized variables, the use of potentially dangerous functions etc. In order to statically detect these vulnerabilities the *AEGIS* derives the abstract model of the program for verification. The free usage of the analyzer is available via the official Digitek Labs web-site [9]. Before running this tool it is necessary to make some transformations of a given C code for further compiling. For example, in the *AEGIS*, it is prohibited to analyze a code where two or more C instructions are located in the same program line. We have correspondingly changed the above Programs 1, 2, 3 and have run the *AEGIS*. For Program 1 of average value calculating the *AEGIS* detected an incorrect array index for the array `arr` while for Programs 2 and 3 of array sorts the *AEGIS* mentioned only the call of unsafe function `system()`.

**F.** There are other static code analyzers that can be used for vulnerability detection in C programs. For example, *Cqual* [10], developed by Dan Wilkerson in 2004, *Eshelon AK-VS* [11] developed in Russia, *Klocwork Truepath* [12] developed by *Klocwork* company and *Coverity Static Analysis* [13] developed by *Coverity* company in USA, *MOPS* [14] and *BOON* [15] are tools for static detection vulnerabilities. We could not execute these tools due to some reasons such as a high price, lack of documentation, absence of demonstrating version etc. However, according to their descriptions [10–15], all these tools are developed for static detection of vulnerabilities and many of them allow static analysis not only for C/C++ code but also for Java or C# programs.

According to the above short survey of static code analyzers, one can conclude that most existing tools only search for dangerous functions and despite of their descriptions do not detect type overflow and incorrect array index vulnerabilities. The latter means that for some kinds of software vulnerabilities static detection is not enough, that is the reason why in the next section we present a brief overview of an approach for dynamic detection vulnerabilities [3].

#### IV. SPIN BASED APPROACH FOR DETECTING VULNERABILITIES

Most existing tools providing dynamic detection vulnerabilities are based on randomly generated input data for a given program. Thus, it is difficult to guarantee the fault coverage for such security testing. There also exist special tools for distributed programs testing, for example, *Helgrind* [16] that is designed for multithreaded programs testing. We note that this tool does not support buffer overflow detection technique but it is able to control synchronization between threads.

There are other model checking techniques which are widely used for vulnerability detection. Working together with our French colleagues we proposed a detection technique based on *SPIN* model checker [1]<sup>2</sup> and have partially presented the obtained results in the technical report [3]. In this case, a vulnerability is described as a property that has to be verified. However, *SPIN* accepts a program written in *PROMELA* language and thus, the first question is how to translate a C code into *PROMELA* instructions when verifying a property of interest. If the program is vulnerable, i.e., possesses a “bad” feature, then *SPIN* produces a counterexample that corresponds to the values of internal variables or of input data of the program. We note that, according to *SPIN* documentation features might be specified as temporal logic formulas or Buchi automata [17]. In the former case, we propose how to inject such data into the program in order to show a programmer which part of the code is vulnerable. The proposed technique somehow takes into account both static and dynamic vulnerability detection, since *PROMELA* model is verified statically while counterexample is injected into the program through its runtime. In [3], some discussions can be found how to translate C instructions into *PROMELA* instructions and how the

<sup>2</sup> The work was done together with French scientific group of Prof. Ana Cavalli (TELECOM & Management Sud Paris)

injection procedure can be implemented. In PROMELA language verified properties are described as assertions and such assertions have to be constructed for each type of vulnerabilities. Unfortunately the translation performed by MODEX tool [18] cannot be applied directly and since we are in the process of developing new automatic tools for such translation, some C codes were manually converted into PROMELA codes and corresponding assertions were added. We have applied a proposed technique to the above Programs 1, 2, 3 and SPIN produced counterexamples for all of them. We injected data according to these counterexamples, found out that the programs return wrong results and no error message about “bad” input data has appeared, i.e., SPIN has detected type overflow and array overflow vulnerabilities in the above programs. For example, for Program 1 a counterexample produced by SPIN has the value 10005 for each array item value, the returned result when running the program was 3451 while the right value should be 100050, i.e., this C code has a type overflow vulnerability.

For Program 2 SPIN produced a counterexample as well as for the array dimension as for array item value. In this case when detecting array overflow vulnerability the counterexample was  $n = 11$  when each array item equals 11 too. When detecting type overflow vulnerability SPIN produced the value 70035 that was then assigned to each array item. After applying these input data to Program 2 incorrect result has been obtained when running the C program while no error occurred. According to the incorrect result that can easily be checked, one can conclude that SPIN has detected type and array overflow in Program 2. For Program 3 (Table 1) SPIN has produced the same counterexample  $n = 11$  for an array overflow while in the counterexample for a type overflow vulnerability, each array item was assigned to 80040.

In order to compare SPIN based vulnerability detection technique with other tools providing dynamic vulnerability detection we have run the *Memcheck* utility of Valgrind software [15] against Programs 1, 2, 3. *Memcheck* is designed to detect memory leakages in C/C++ programs and incorrect use of uninitialized values. Valgrind allows a programmer to assign desirable values to input variables and by use of a virtual machine the *Memcheck* utility checks whether memory leakage occurs during the program execution. We have run *Memcheck* against Programs 1, 2, 3 with counterexamples produced by SPIN and neither type overflow nor array overflow vulnerability has been mentioned.

Based on the obtained experimental results, we can conclude that SPIN based detection techniques could be useful when analyzing the C code safety.

## V. CONCLUSIONS

In this paper, we have presented a short survey of existing tools providing vulnerability detection in C/C++ programs. Several tools have been executed against student implementations of array algorithms. The experimental results clearly show that for some kinds of C code vulnerabilities static analysis can be insufficient and we have presented a brief overview of a SPIN-based approach for vulnerability

detection. The obtained preliminary results clearly show that SPIN based detection techniques could be useful when analyzing the C code safety. In this paper, we did not discuss vulnerability detection techniques based on other model checkers; such a comparison is a part of our future work.

## REFERENCES

- [1] G. Holzmann. Spin Model Checker. Primer and Reference Manual. Addison Wesley, 2003.
- [2] Willy Jimenez, Amel Mammar, and Ana R. Cavalli. Software Vulnerabilities, Prevention and Detection Methods. A Review, SEC-MDA workshop.– Enschede, The Netherlands, June, 24, 2009.
- [3] Technical report of the joint FCP Russian-French grant № 02.514.12.4002, Step 4.
- [4] Cigital [Electronic resource] – <http://www.cigital.com/its4/>
- [5] Flawfinder home page [Electronic resource] – <http://www.dwheeler.com/flawfinder>
- [6] Just Another Hacker [Electronic resource] – <http://www.justanotherhacker.com/projects/graudit/download.html>
- [7] Sound Forge [Electronic resource] – <http://sourceforge.net/apps/mediawiki/cppcheck/>
- [8] Digitek Labs [Electronic resource] – <http://www.digiteklabs.ru/aegis/>
- [9] Digitek Labs [Electronic resource] – <http://aegis-demo.digiteklabs.ru/s2a.websserver/>
- [10] Department of computer science. University of Maryland [Electronic resource] – <http://www.cs.umd.edu/~jfoster/cqual/>
- [11] Soft Line [Electronic resource] – <http://soft.softline.ru/NPO-Echelon/eshelon-ak-vs/>
- [12] Klocwork [Electronic resource] – <http://www.klocwork.com/products/insight/klocwork-truepath/>
- [13] Coverity [Electronic resource] – <http://www.coverity.com/products/static-analysis.html>
- [14] Electrical engineering and computer sciences [Electronic resource] – <http://www.cs.berkeley.edu/~daw/mops/>
- [15] Electrical engineering and computer sciences [Electronic resource] – <http://www.cs.berkeley.edu/~daw/boon/>
- [16] Valgrind [Electronic resource] – <http://valgrind.org/info/tools.html>
- [17] SPIN [Electronic resource] – <http://spinroot.com/>
- [18] Modex [Electronic resource] – <http://cm.bell-labs.com/cm/cs/what/modex/index.html>



# Model checking approach to the correctness proof of complex systems

Marina Alekseeva

P.G. Demidov Yaroslavl State University  
150000 Yaroslavl, Sovetskaya 14, Russia  
Email: marya\_87@mail.ru

Ekaterina Dashkova

P.G. Demidov Yaroslavl State University  
150000 Yaroslavl, Sovetskaya 14, Russia  
Email: dea.yar@mail.ru

**Abstract**—Very often the question of efficiency in terms of execution time memory usage, or power consumption of the dedicated hardware/software systems is of utmost interest that is why different variants of algorithms are developed. In many situations the original algorithm is modified to improve its efficiency in terms like power consumption or memory consumption which were not in the focus of the original algorithm. For all this modifications it is crucial that functionality and correctness of the original algorithm is preserved [1].

A lot of systems increasingly applying embedded software solutions to gain flexibility and cost-efficiency. One of the various approaches toward the correctness of systems is a formal verification technique which allows to verify the desirable behavior properties of a given system. This technique nowadays is well known as model checking. Model is expected to satisfy desirable properties.

Verification is the analysis of properties of all admissible program results through formal evidence for the presence of required properties. The basic idea of verifying the program is to formally prove the correspondence between the programming language and the specification of the problem.

Program and specification describe the same problem using different languages. Specification languages are purely declarative, human-centered. Imperative programming languages are more focused on executing on the computing device. Therefore less natural for men.

Likewise, this technique is an excellent debugging instrument. From the standpoint of programming technology verification enables to obtain a better strategy for debugging programs.

**Index Terms**—verification, automata-based programming, complex systems.

## I. INTRODUCTION

Correctness of Information and Communication Technology (ICT) systems [2] is the background for their safety. Errors could be catastrophic. The fatal defects in the control software are very dangerous and the number of defects grows exponentially with the number of interacting system components. Day after day ICT systems are becoming more complex.

ICT systems are universal and their reliability is the main point in the system design process. The key instrument for design process is verification techniques (fig.1). The features which are verified could be taken from specification. They are usually the main properties of the systems. They should be correct which means react adequate for any command. The accurate modelling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications.

Such problems are usually discovered at later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques as model checking.

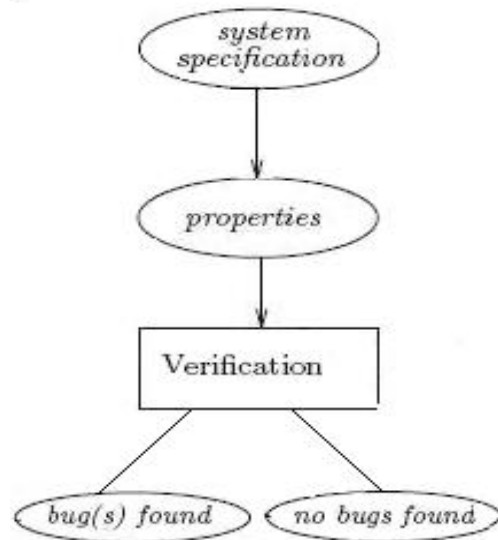


Fig. 1. The process of verification

## II. MAIN PART

### A. Model-checking

Model checking [3] is one of various verification techniques. It explores all possible system states in a rude manner.

The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming or hardware description languages.

The property specification prescribes how the system behaves. All relevant system states are checked whether they satisfy the desirable property or not (fig.2).

Models of systems describe the behavior of systems in an accurate and unambiguous way. They are mostly expressed using finite-state automaton, consisting of a finite set of states and a set of transitions. In order to improve the quality of the

model, a simulation prior to the model checking can take place. Simulation can be used effectively to get rid of the simpler category of modelling errors. Eliminating these simple errors before any form of thorough checking takes place may reduce the costly and time-consuming verification effort.

Model checking has been successfully applied to several ICT systems.

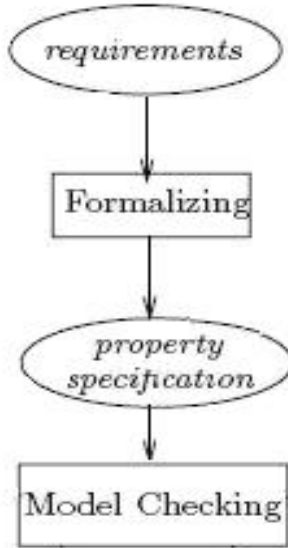


Fig. 2. The process of model-checking

### B. Automata-based programming.

Automata-based programming can be used in several types of programming systems [4]:

- transforming systems (compilers, archivators). Finite automaton in programming traditionally used in design of compilers. In this situation automaton is understood as some calculating feature which has an input line and output line.

- reactive systems (telecommunication systems and systems of control and managing of physical devices). In this case the automata-based programming solves the problem of logic programming. Automaton is a device that has several parallel input lines (often binary), on which in real time the signals from the environment are coming. Processing such kind of signals, automaton is forming values for several parallel outputs.

So, the usefulness of the automata-based approach can be characterized with the combination of the words "complex behavior". For such kind of systems it is very important that automata-based approach separates the description of logic of behavior and semantics. This feature makes automaton description of complex behavior clear and understandable.

Transition systems are often used in computer science (semantical models for a broad range of high-level formalisms for concurrent systems, such as process algebras, Petri Nets, statecharts). They are a fundamental model for modelling software and hardware systems.

Transition system is defined as  $TS$ .  $TS$  is a tuple  $(S, Act, \rightarrow, I, AP, L)$  where

- $S$  is a set of states,
- $Act$  is a set of actions,
- $\rightarrow \subseteq S \times Act \times S$  is a transition relation,
- $I \subseteq S$  is a set of initial states,
- $AP$  is a set of atomic propositions, and
- $L : S \rightarrow 2^{AP}$  is a labeling function.

$TS$  is called finite if  $S$ ,  $Act$ , and  $AP$  are finite.

Consider the following example (fig.3). The transition system in fig.3 is a schematic design of an automaton. The automaton can either deliver tea or coffee. States are represented by ovals and transitions by labeled edges. Initial states are arrow without source.

The state space is

$$S = \{pay, select, tea, coffee\}.$$

The set of initial states consists of only one state, i.e.,  $I = \{pay\}$ .

The action insert coin denotes the insertion of a coin, while the automaton actions get tea and get coffee denote the delivery of tea and coffee. Transitions of which the action label is not of further interest here are all denoted by the distinguished action symbol  $\tau$ . We have:

$$Act = \{insert\_coin, get\_tea, get\_coffee, \tau\}.$$

Automaton is represented by two locations pay (start) and select. Notes that after the insertion of a coin, the automaton nondeterministically choose to provide either coffee or tea.

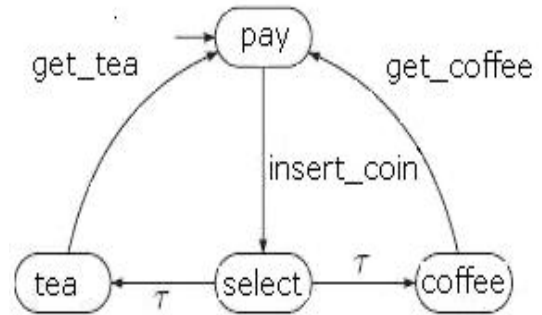


Fig. 3. A simple transition system

## III. RESULTS

Authors had an experience of applying the model checking method. Their diploma paper was devoted to the verification of the WTP (Wireless Transaction Protocol). The simple transactions were built with the help of CPN Tools and NS2 Simulator. Two types of instruments were explored.

### A. System modeling. NS2.

Simulation is widely-used in system modeling for applications ranging from engineering research, business analysis, manufacturing planning, and biological science experimentation. Network Simulator (Version 2), widely known as NS2, is an event driven simulation tool which is very useful in

studying the dynamic nature of communication networks. NS2 provides users with a way of specifying such network protocols and simulating their behaviors. NS2 suggest two steps of work. The first step is constructing a model with the help of programming on C++, and finally the use of the Object-oriented Tool Command Language (OTcl) for analysis of the model and simulating the network conditions. It allows us to include our C++ programming code to the NS2 environment. We decided that NS2 is the most convenient tool for modeling the network behavior.

### B. Proposed model.

The Wireless Transaction Protocol is responsible for reliable message delivery. Maximum Transfer Unit (MTU) is a maximum size of a packet in networks. If we have a message that is bigger than MTU then WTP fragmentizes this message. Flow control in cases of fragmented messages, is performed by sending fragments in groups. Every group of packets requires only one acknowledgement of the group. The last packet of each group contains a special flag. This flag indicates the end of the group and receiver knows when to send an acknowledgment. Size of each group depends on the link characteristics and the device memory. It is necessary to avoid extra packet retransmission and data loss. Receiver sends a negative acknowledgement (NAK) if the end-of-group packet is received whilst intermediate packets are missing. This operation is repeated until the entire group is received and a positive acknowledgment is sent. If timeout occurs, only the last packet of the group is retransmitted, and sender knows what packets have been lost. Wireless Transaction Protocol tries to minimize the number of unnecessary retransmissions.

In our model we have three parameters:

- $t_s$  is the time interval between consecutive packets of the group which are sent from the sender SENDER to the receiver RECEIVER.
- $t_r$  is the interval between consecutive packets of the group which are received by the RECEIVER.
- $P_{am}$  as the number of packets in the group.

In our model there are two types of acknowledgments (ACK is a positive and NAK - negative acknowledgment).

When receiver sends an acknowledgment it transfers  $t_r$  with the help of it. Sender calculates special ratio. Depending on the result of this ratio sender has several situations for analysis and further actions.

- Perfect network conditions.
- Parameters can be modified by increasing  $P_{am}$ , decreasing  $t_s$  and timeout.
- There is no enough data for our algorithm to make a decision how to modify parameters (conditions of a network correspond to the established parameters).
- The network is congested, parameters can be modified by decreasing  $P_{am}$ , increasing  $t_s$  and timeout.

## IV. CONCLUSION

Theory of programming even in the 1968 openly accepted the crisis of software development. The main symptom of

this crisis is disability of the developers to provide the main feature of the software: its correctness. Theoreticians and practitioners of software underline that the crisis of methods of the development of software shows mainly during the design of the systems with complex behavior and automata-based approach can deal with this problem. That is why it is the answer for the most up-to date problems of the software development industry. The predictions show [4] that the area of applying automata-based programming will be expanded and this technology will be developed. A new models, notations and instruments will appear in the foreseeable future.

## ACKNOWLEDGMENT

The following scientific advisers supported us by using (sometimes very) preliminary versions of this article: Valery A. Sokolov (Yaroslavl, Russia), Dmitry U. Chaly (Yaroslavl, Russia), Egor V. Kuzmin (Yaroslavl, Russia).

The authors would also like to thank the dean of Yaroslavl Demidov State University Computer Science Department P.G. Parfenov for interest and support of this project and the head of scientific-educational center "Center of Innovation Programming" Professor V.A. Sokolov for helpful advices. This work would be developed and extended in the future.

## REFERENCES

- [1] Anikeev M., Madlener F., Schlosser A., Huss S.A., Walter C., "Automated Correctness Proof of Algorithm Variants in Elliptic Curve Cryptography" *Modeling and Analysis of Information Systems*, pp. 7–16, 2010.
- [2] Baier Christel, Katoen Joost-Pieter. "Principles of Model Checking," *The MIT Press, Cambridge, Massachusetts, London, England*, 2008.
- [3] Egor V. Kuzmin, "Introduction to the theory of mathematical processes and structures," *Yaroslavl Demidov State University, Yaroslavl, Russia*, 2001.
- [4] N.I. Polikarpova, A.A. Shalyto, "Automata-based programming" *Saint-Petersburg State University of Informatic Technologies, Mechanics and Optic, Saint-Petersburg, Russia*, 2009.

# Thorn language: a flexible tool for code generation

Yuri Okulovsky

Ural State University  
Yekaterinburg, Lenina str. 51  
Email: yuri.okulovsky@gmail.com

**Abstract**—This paper presents a new approach to domain-specific languages creation. Instead of defining both language syntax and semantics for each case, the same general-purposed markup language Thorn is used. The original model of translation associates commands inside a Thorn document with programs written in some script language. When the compiler needs to execute a command, it launches a corresponding program, passes data from the document to the program, and uses an output value of the program as the result of the command. We describe an approach to code generation based on Thorn, and compare the approach to other known code generation methods. We give various examples of Thorn-based code generators.

**Index Terms**—code generation, domain-specific languages, automated programming, language-oriented approach

## I. INTRODUCTION

Every programmer likes writing elegant code, implementing sophisticated algorithms and developing original architecture. Real software, however, often consists of stereotyped and uncreative code: business logic control, layout of widgets, interaction with database, automata dispatch tables, etc. The more features a software product has, the larger amount of stereotyped code it contains. This leads to the staff expansion and project management issues. In addition, a comprehensive testing of stereotyped code is required. The natural desire emerges to eliminate these inconveniences.

In fact, all the history of programming languages is a history of attempts to decrease the percentage of stereotyped code and to improve its structure. Simple arithmetical operations in Assembler require several lines of code. More modern languages like C allow writing the same operations in one line. Memory management in C demands special attention, leading to lots of stereotyped code and numerous memory leaks. These issues was resolved with garbage collection in Java and C#. The most recent developements in programming languages simplify significantly collection management and interaction with databases (LINQ [13]), widgets layout (WPF [13]), or contracts control (Eifell [10]). However, software products often have unique patterns in addition to well-known ones. These patterns are typically not supported by languages.

The promising approach to described problems is code generation: writing programs that write programs [8], [5]. The simplest case is a tool that allows description of the desired code in text or graphic format, and then produces the code. For example, Microsoft Visual Studio contains a special wizard for visual creation of widgets and forms [2]. Another example is production of parsers from grammars [9]. There are many

code generators, which produce specific software from UML diagrams or other description of data [7]. Code generator tools cover areas that are less common than database handling or widgets layout. They are not, however, a general solution of the stereotyped code problem. There is no general approach behind these tools, and so they cannot be reused to produce code for other patterns.

Domain-specific languages (DSL, [14]) are an attempt to apply code generation to even narrower areas. In the DSL approach, a new small language is created for every specific code pattern. The language is applicable only in its domain: it is not general purpose language, and may even not be Turing-complete. DSL are widely used in program engineering.

The obvious way to create DSL is to define its grammar, then write or generate a parser and implement a translation scheme on certain program language. Language analysis requires substantial additional competence of a programmer and therefore sets an “entry threshold“ that limits the code generation availability. In many cases, it is simpler just to write a templated code manually than to design a new language. There also exist several tools for DSL creation, the most prominent being Visual Studio DSL Tools [4] or JetBrains Meta-Programming system [6]. These tools have immense capabilities and allow creation of very complex models and languages. However, studying these tools is even more complex than writing DSL compiler manually.

Our concern is to decrease a complexity of DSL creation and therefore improve a code generation availability. We want to make code generation so simple that its usage would be reasonable even in simplest cases. We explore a different approach to domain-specific languages. By our observations, domain-specific *language* is not required in many cases. We may use the same general-purpose markup language (like XML) in different domains to describe a desired program code. However, domain-specific *semantics* is still needed, because the way the description is to be processed changes with each domain. We propose a new way to create code generators: a language with separted semantics. The language only defines how we should markup the data with commands, and does not specify what the commands mean. The logic of commands is written in arbitrary programming language for each domain.

Our approach to code generators resembles the Common Gate Interface (CGI) approach to web applications. CGI does not require invention of a special language for each web site, like the DSL approach. It does not make us write

individual programs that analyze HTTP packages. Instead, CGI parses packages itself and adopts environment variables and standard input to carry information from the package. Then CGI launches an arbitrary program, which processes the request, accepts an output of the program and sends it back to the user over the network. In spite of thorough research, we have not found this model to be implemented in any known programming language or tool.

Our approach to code generation was implemented in Thorn — a new programming language with separated semantics, designed especially for code generation. It is an Open source product under GPL v3 licence. Thorn is a fully operable software product, tested for more than four years, with successful practice in code generation.

In the first section, we describe Thorn language: the syntax of Thorn document, the way to create a new command and details of the way commands are executed. In section 2 we describe a relatively simple code generator for HTML and WordprocessingML [1] documents. In section 3 we describe the Thorn programming technique for creation of complex generators — functional generators. We also describe two examples of functional generators. The first generator converts bibliographic data from Thorn to BibTeX, HTML, WordprocessingML and some other formats. The second generator, described in section 4, is rather a programming framework that consists of Thorn libraries and C# auxiliary assemblies.

## II. THORN ESSENTIALS

Thorn document is a tree of nested commands. The example of Thorn syntax is shown in Listing 1. We may compare the Thorn document with the corresponding HTML document which is shown in Listing 2.

**Listing 1** Thorn code for a table.

```
\table[border=1 align=center] {
  \tr {
    \td {a11} \td {a12}
  }
  \tr {
    \td {a21} \td {a22}
  }
}
```

**Listing 2** HTML code for a table.

```
<table border=1 align=center>
  <tr>
    <td> a11 </td> <td> a12 </td>
  </tr><tr>
    <td> a21 </td> <td> a22 </td>
  </tr>
</table>
```

Thorn document is more compact than HTML. Its size may be further decreased. Firstly, if the order of param-

eters of a command is fixed, their names may be omitted: `\table[1 center]`. Secondly, we may define the nesting of commands. We know `\tr` command is always inside `\table`, and `\td` is inside `\tr`. Therefore, curly brackets may also be omitted. The resulting code is shown in Listing 3.

**Listing 3** Compact Thorn code for a table.

```
\table[1 center]
  \tr \td a11 \td a12
  \tr \td a21 \td a22
```

Parsing of input stream is provided by pushdown automaton. It should be mentioned that due to curly brackets omitting, Thorn grammar is ambiguous. Depending on commands' definitions, the string `\cmd1 \cmd2` may be interpreted as `\cmd1{\cmd2{}}` or `\cmd1{\cmd2{}}`, which implies different parse trees.

The order of parameters' names, the rules of nesting and the logic of commands are defined in command files. The set of such files (library) must be loaded by Thorn before document compilation starts. The example of `\table` command file is shown in Listing 4.

**Listing 4** Declaration of `\table` command.

```
#Keys=border,align;Blocks=entry;
#Parents=body;Type=Perl;Free=yes;

$STRING="<table border=$PARAM{border}
          align=$PARAM{align}>
  $TEXT{entry}
</table>";
```

The command description starts with service section, which is marked with `#` symbol (lines 1-2 in Listing 4). The service section specifies the default order of two parameters, `border` and `align`. Their names may be omitted, as in Listing 3. The command description also specifies the name of text entry to use inside the program. Other parameters specify the command, which can contain `\table`, the language the command logic is programmed in, the fact that we may omit curly brackets for `table` command.

After the service section is completed, the program in Perl is written. This program fills `$STRING` variable, which is an output of the program. It uses special hashes `%PARAM` and `%TEXT`, which store text variables from the input document. Aside from special variables, the program is arbitrary: it can manipulate files, use modules, etc. The program is executed not by Thorn, but by the Perl compiler. When Thorn acquires all the information about the command `\table` (in Listings 1 and 3, these are values '1' and 'center' and the result of execution of two `tr` commands), it launches the Perl compiler with 'processor' program, and stores information about command and its parameters to `STDIN`. The processor reads the information from `STDIN`, executes the command code by `eval` function, and prints the result to `STDOUT`.

Thorn compiler reads STDOUT, removes the command from the document and places the result in corresponding place.

Commands may return an error message, which will be passed to the user. Commands may interact by global variables, stored in %GLOBAL hash. Processor obtains %GLOBAL hash from Thorn compiler and sends it back each time. Macros are also available: the result of a macro command will be processed by Thorn again.

Currently, there are three ways to execute the Thorn command:

- Use Perl commands;
- Make the new type of command and write a plugin that executes this type. This includes addition of new script languages in Thorn;
- Reference Thorn.dll in a .NET project, associate a command name with a class that implements corresponded interface, and launch Thorn compiler from the project.

Commands can be documented in Thorn language. A comment section is placed before a command section with ## marker. It contains a commentary in Thorn language with special commands like \desc (general description), \key[i] or block[i] (i-th parameter or text block description), etc.

Let us give an example of basic code generation of C# code. Consider a command \event in Listing 5. This command

---

**Listing 5** Command for generation of events.

---

```
#Type=Perl; Keys=Type,Name; Blocks=Comment;
$type=$PARAM{Type}EventHandler;
$name=$PARAM{Name}EventSize;
$args=$PARAM{Type}EventArgs;

$STRING.="
///<summary>
///$TEXT{Comment}
///</summary>
public event $type $name;

///<summary>Raises $name</summary>
protected virtual void On$name($args e) {
    if ($name!=null) $name(this,e);
};
```

---

will transform the following Thorn code:

```
\event[Mouse MouseMove]{Comment}
```

into a declaration of event and corresponding invocation method.

Several simple libraries for generation of C# code have been developed. Commands in these libraries can generate properties (with custom access modifiers and optional invocation of event), events, enumerations, switch operators (in case they are large and nested) and other templates for fast C# programming. We can use commands from different libraries in the same document. It corresponds to merging code generator applications, but does not require any special efforts.

**Comparison to XML and XSL approach.** In certain degree, Thorn follows XML/XSLT approach to HTML/CSS generation. In this approach, data is written in an XML document. The document is then converted into HTML with XSLT. It is even possible to use XSL to convert XML document into Java source code [3].

We argue that Thorn is more comfortable for code generation than XML/XSLT technology. The key difference is a possibility to use an arbitrary language for commands' logic. That simplifies generators greatly. Note how natural and readable listing 5 looks, especially in comparison with XSLT schemata in [3]. We may develop different commands in different languages, therefore choosing the most fitting language. In addition, we may create new command types and therefore patterns for commands' logic, as it will be shown below.

It is possible to develop an XML compiler that acts exactly as Thorn compiler: parses tags and processes them with Perl. The reason why we have developed a new language is that we wanted to minimize manual typing and therefore make Thorn more comfortable for code generation. Still, Thorn approach to compilation can be applied to other languages.

**Comparison to DSL approach.** Thorn language itself is not domain-specific, since its semantics is not defined. However, Thorn with selected set of libraries can be considered as a domain-specific language. Therefore, Thorn can be viewed as a tool for DSL creation. We believe that Thorn is much simpler than other such tools. There is no need to describe tokens, write down language grammar, etc. To create language semantics we only need to write simple Perl programs, and demands for these programs to be Thorn commands are not burdensome. Summarizing, Thorn has a very low "entry threshold" and can be used for fast creation of small DSL for a project, hence making code generation more available.

Thorn can be also used as a back-end for a compiler. Commands of Thorn form a tree, therefore a parse tree of a front-end compiler can be stored as a Thorn document and then interpreted with appropriate library.

### III. CODE GENERATION FOR MARKUP LANGUAGES

In this section, we consider generation of documents written in markup languages [11]. We have developed a library for producing an HTML code from Thorn description. It supports all HTML tags. It defines nesting of tags so curly brackets are rarely used. It specifies orders of most popular parameters of tags. In addition, if a parameter name starts with ! symbol (!SomeStyle), it will be placed in style attribute as SomeStyle=Something. All these improvements make Thorn files very small and readable in comparison with generated HTML.

All commands for HTML tags perform the same logic. The name of a command is translated into a tag, all attributes are listed after this tag with their names, the only text block is placed inside a tag, etc. Hence, we actually do not need to program the command logic in Perl, as in Listing 4. Instead, we develop new command types. In command declaration,

we write `Type=HTMLPairedTag` instead of `Type=Perl`. Commands of this type are executed by the Thorn compiler itself, without launching the Perl compiler. It improves performance greatly, since the major time of Thorn work lies in passing parameters between the compilers.

The second library is a library for producing `WordprocessingML` [1]. `WordprocessingML` is an XML dialect for text processors. This standard is supported by Microsoft Word, Open Office Word Processor and other text processors. Output in `WordprocessingML` format allows using all features of text processors. However, `WordprocessingML` files are not easy to type. The first reason is XML being redundant. The second reason is that `WordprocessingML` reflects the logic of a word processor, but not of a human. For example, items of multi-level lists are not really nested within each other, as in HTML or TeX. Instead, each item is a paragraph, and its level is determined by a style. Bold and italic words are not embedded in a plain text. Instead, the plain text ends, then a text with bold style starts and ends, and then plain text continues. There are many other similar inconveniences. We have developed the Thorn library, which allows to write Thorn documents in a habitual way (very much like TeX) and then to transform them into `WordprocessingML`. Not all features of `WordprocessingML` are currently supported.

A special extension for both these libraries (and potentially for any library that produces text documents) is created. This extension allows to create not only a document, but also a program that produces this document. Consider the code in Listing 6.

---

**Listing 6** A document with a variable inside.

---

```
\document \html \body

\p Variable equals
  \variable[name=Var type=Int default=5]
```

---

Commands `\document` and `\variable` are defined in two libraries: `lib.programmer` and `lib.makerup`. In `lib.makerup`, `\document` does nothing, it only returns its entry. The command `\variable` returns default value ('5' in Listing 6). Therefore, the maker-up sees an example of a document, which will be created by a program. In `lib.programmer`, these commands are defined differently. Command `\variable` is transformed into a marker, which separates the text into variable and invariable parts. Command `\document` assembles parts and produces methods, which take all variables as arguments, and store a resulting document in a stream. There are two versions of `lib.programmer`: PHP version for web sites and C# version for offline software.

#### IV. FUNCTIONAL GENERATORS

When a code generator becomes more complicated, parameters of each command become numerous and hard to remember. We need to divide one command into several. This can be done by using relations. The relation is a set of records.

In each record, some fields are specified. Instead of producing output, Thorn commands fill a relation in global variables. In many cases, a generator that translates a relation description into source code can be represented as a following function (which we call functional generator):

$$p(A) = h( \begin{matrix} g_1[ f_1(A_1), \dots, f_1(A_m) ] \\ g_2[ f_2(A_1), \dots, f_2(A_m) ] \\ \dots \\ g_n[ f_n(A_1), \dots, f_n(A_m) ] \end{matrix} ).$$

Here  $h$  produces the source code. In an object-oriented program,  $h$  typically produces one class. Functions  $g_1, \dots, g_n$  produce parts of code, for example, methods inside the class. Functions  $f_1, \dots, f_n$  produce parts of methods, corresponding to one record in the relation.  $A_1, \dots, A_m$  are records of the relation. Usually,  $g_i$  are concatenation functions, i.e.  $g_i(x_1, \dots, x_n) = x_1 \cdot \dots \cdot x_n$ , where  $\cdot$  denotes the concatenation operation. Also,  $h$  can usually be represented as

$$h(x_1, x_2, \dots, x_n) = a_0 \cdot x_1 \cdot a_1 \cdot x_2 \cdot a_3 \cdot \dots \cdot a_n \cdot x_n \cdot a_{n+1},$$

where  $a_i$  are string constants.

A simple way of representing relations in global variables is chosen. A value of field `Field` in a record with number `N` in a relation `Relation` is stored in global variable `Relation#N#Field`. This representation is supported by a new type of Thorn commands, as with HTML commands. We have also developed a Perl module that provides user-friendly way to access relation in global variables. A set of methods to implement functional generators in Thorn and Perl is called `Fungi` (functional generator interface).

Based on `Fungi`, we have developed `BibThorn`, an analogy to `BibTeX`. `BibTeX` is a flexible and widespread technology for storing bibliographic data. However, TeX can generate only a small set of formats. TeX cannot produce HTML files to place bibliography on a web site, or a plain text to include in a scientific report. Using `Fungi`, we may describe bibliography as a relation. A simplified example is placed in Listing 7.

---

**Listing 7** A bibliography information on Thorn.

---

```
\bibliography
  \item[book]
    \author John Smith
    \title My book
  ...
\print
```

---

Command `\item` adds a new record in a relation. Commands `\author` and `\title` fill corresponding fields. The logic of `\author` command could be encoded on Perl as in Listing 8.

Module `db.pm` is a `Fungi` implementation for Perl. It allows easy access to relations in global variables. Since such commands are required for authors, title, publisher and other fields, their logic is stereotyped. Command `\author` may therefore be described as in Listing 9.

---

**Listing 8** Perl implementation of `\author`

---

```
#Blocks=Author;Free=yes;
#Parents=item;Type=Perl;
require 'db.pm';
$db=db->new(\%GLOBAL);
$db->SetFieldInCurrentRow
    ("Authors", $TEXT{Author});
```

---

---

**Listing 9** Fungi implementation of `\author`

---

```
#Blocks=Author;Free=yes;
#Parents=item;Type=FungiSetter;
```

---

Command `\print` actually prints the bibliography. The prototype of the `\print` command is shown in Listing 10.

---

**Listing 10** Using Fungi in Perl command to create a bibliography.

---

```
#Type=Perl;
require 'db.pm';
sub MakeItem {
    %h=@_;
    $STRING.=" $h{Author}. $h{Name}\n";
}
$db=db->new(\%GLOBAL);
$db->RunOver(\&MakeItem);
```

---

`RunOver` method looks through records, copies each record into a hash, launches `MakeItem` method and passes the hash to the method.

## V. THORNADO FRAMEWORK

Thornado is a framework that allows fast creation of business software [12]. The main aim of Thornado is assistance in input and output of data. It contains a code generator for data description, and a .NET library with useful templates. Listing 11 demonstrates a description of one field.

---

**Listing 11** Description of one field in the extended system for data description.

---

```
\field[string Email]
\desc E-mail address
\io TypeIO.String
\gui Text
\check
\err.W v=="
    -- Address must be entered
\err.W v.Length<5
    -- Address is too short
if (!v.Contains('@'))
    list.Add("Address is not valid");
```

---

Command `\field` specifies the type of the field and its name. Command `\desc` specifies the commentary for this field in the generated code. It also specifies the caption for this field in graphic user interfaces.

Command `\io` specifies an object that transforms a field into a string and parses it from string. In C#, primitive types (`int`, `double`, etc.) can be written and parsed from a string by .NET means. Unfortunately, many types do not support parsing and writing in human-readable forms. The way to specify how exactly the object should be converted is often entangled and inconvenient. There is also no way to read and write null value. Therefore, we introduce `TypeIO` classes for input and output objects. For example, methods `TypeIO.Int.Write` and `TypeIO.Int.Parse` convert `int` value into a string and vice versa. `TypeIO.Int` is a predefined object of `IntIO` class. Many formats for the same type may exist: for example, `double` can be processed with `TypeIO.Double`, `TypeIO.Double.Money` or `TypeIO.Double.Percent`. Many `TypeIO` classes are developed, including those for classes that are not usually converted to string (like colors, pens and brushes). In addition, each `TypeIO` object `X` has properties `X.Nullable` that may process null and `X.InArray` that may process arrays.

Command `\gui` specifies a type of graphic user interface. Several types are available. `TextBox` option allows input of an arbitrary string with following conversion into a value with specified `TypeIO` object. `ListBox` means selecting one value from a list. In this case, `\values` command allows to specify items of the list, which is arbitrary `IEnumerable` object. Note that it is impossible to specify an object as an attribute in C#.

Finally, `\check` command describes the business logic of this field. The logic can be described by Thorn commands (`\err`, for example). It is also possible to place pure C# code into `\check` command with predefined variables `v` (a new value of the field) and `list` (a list that stores errors). This code will be placed in corresponding location of a generated source code. Business logic for one field is to be inserted into corresponding property. Business logic for the whole class is to be placed into a method `CheckConsistency`. The method is called before input-output procedure begins and after it has been completed.

Thornado can generate a class with required fields and business logic. Each field of generated class is associated with `FieldInfo` object. `FieldInfo` contains all the information we specify in Thorn file: caption for a field, a type of graphic user interface, etc. Thorn can generate `IOPProvider`, which stores a collection of `FieldInfo` objects and can manipulate fields of generated class. Other classes perform various operations by using `IOPProvider`: input and output into INI- and XML-files, interconnection with ADO.NET, generating of graphic user interface widgets, etc. Therefore, we can generate a substantial part of an application from Thorn description.

`IOPProvider` performs some sort of reflection. It is more convenient than traditional C# reflection. Names of



FieldInfo objects are C# variables and are checked in compile time, unlike string values that are used in reflection. FieldInfo objects carry all additional information in their fields, and there is no need to read attributes to access them. Finally, it is impossible to write in attributes neither the arbitrary business logic nor the references to other classes and objects.

## VI. CONCLUSION AND FUTURE WORKS

In this paper, we presented Thorn language, studied its differences from other approaches to DSL creation and described its basic programming techniques. Thorn is a fully operable software product, available under GPL v3 licence. Its area of usage is creation of small code generators for simple code patterns. Due to its simplicity, Thorn seems to be more preferable than other tools for DSL creation, or manual writing DSL compilers.

We consider following ways to improve our product:

- \*nix version of Thorn
- Plug-ins for Microsoft Visual Studio, Eclipse and other popular integrated development environments
- More languages will be supported "from the box" for command development
- BibThorn will be extended and integrated with L<sup>A</sup>T<sub>E</sub>X editors
- Thornado framework will be further developed

## REFERENCES

- [1] Standard ecma-376. <http://www.ecma-international.org/publications/standards/Ecma-376.htm>.
- [2] Windows forms designer. [http://msdn.microsoft.com/en-us/library/e06hs424\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/e06hs424(VS.80).aspx).
- [3] E. M. Burke. *Java and XSLT*. O'Reilly, 2001.
- [4] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.
- [5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.
- [6] S. Dmitriev. Language oriented programming: The next programming paradigm. [http://www.jetbrains.com/mps/docs/Language\\_Oriented\\_Programming.pdf](http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf), 2004.
- [7] K. Fertalj and M. Brcic. A source code generator based on uml specification. *International journal on computers and communications*, 2(1), 2008.
- [8] J. Harrington. *Code Generation in Action*. Manning, 2003.
- [9] J. Levine. *Flex and Bison. Text processing tools*. O'Reilly Media, 2009.
- [10] B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997.
- [11] Y. Okulovsky and D. Deyev. System of generation of documents in html, mht and wordprocessingml formats (russian). *Bulletin of Saint Petersburg State University of Information Technologies, Mechanics and Optics: Mechatronics, Technologies and Computer-aided design*, (70), 2010.
- [12] Y. Okulovsky, D. Deyev, V. Popov, and V. Chasovskikh. Code-generation system thornado and its application to creation of business-software (russian). *Bulletin of Saint Petersburg State University of Information Technologies, Mechanics and Optics*, (87), 2008.
- [13] A. Troelsen. *Pro C# 2010 and the .NET 4 Platform*. APress, 2010.
- [14] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

# One Approach to Aspect-Oriented Programming Implementation for the C programming language

Eugene Novikov  
Institute for System Programming  
Russian Academy of Sciences  
Moscow, Russian Federation  
Email: joker@ispras.ru

**Abstract**—The given paper introduces an approach for aspect-oriented programming implementation developing intended for the C programming language. Key features of C and a common C program build process are considered and it's shown how they influence on a supposed C AOP implementation. The last is described in details and after all its practical application is demonstrated. It's shown that the supposed C AOP implementation works well enough although it possesses some shortcomings. Some improvements required to overcome the given shortcomings are discussed.

**Keywords**—aspect-oriented programming; join point; pointcut; advice; aspect; weaving; the C programming language; implementation

## I. INTRODUCTION

Aspect-Oriented Programming (AOP) is a rather fresh programming paradigm that is intended to increase program modularity by means of cross-cutting concerns separation. Generally speaking cross-cutting concerns mean functionality or features that cannot be easily decomposed from so-called core concerns. The last in depend on a programming paradigm used is implemented as corresponding functions, classes and modules while cross-cutting concerns scatter through them and tangle a program source code. The typical example of cross-cutting concerns is logging. Also some more complex fields like errors handling, some sort of testing, security, and database operations can be treated as cross-cutting concerns. AOP provides programmers with opportunity to extract cross-cutting concerns into separate modules called aspects. To understand better let us consider other major AOP conceptions since they'll be widely used at the rest of the given paper.

The basic AOP conception is a **join point**. In general join points are those elements of the programming language semantics which the aspects coordinate with [1]. The given paper takes a join point to be a program construction connected with its context. The typical example of a join point is a function/method call because of such a construction can be found almost in any programming language. But generally speaking join points depend on and even in some degree are determined by a programming language used. A **pointcut** is a set of join points satisfying a given condition. For instance, all memory allocating function (like *malloc*, *calloc* and so on) calls may be treated as a pointcut. Next AOP conception is an **advice**. An advice consists of a pointcut and a body. The last

represents some actions to be executed in matching between a join point corresponding to a given pointcut and a program construction related with some context. Moreover an advice contains information on whether these actions should be executed **before**, instead of (**around**) or **after** a matched program point execution. Usually an advice body is written in a given programming language although some special AOP constructions (e.g. a matched entity name) may be also available. An **aspect** already mentioned above is a separate module that consists of a number of advices implementing some part of cross-cutting concerns. More exactly an aspect also can contain some other constructions, e.g. named pointcuts that is pointcuts associated with identifiers for following usage. At last, the process of aspects with main program integration is referred to as **weaving**. Weaving can be done at any stage of a program processing (at compile time, at post compile time, up to run time) that is exhibited by different approaches.

An AOP implementation depends on a programming language used as was said. Generally an AOP implementation represents a programming language superset required to write aspects and some tool(s) to weave aspects with programs. Let us consider the most advance and popular AOP implementation AspectJ [2] intended for the Java programming language. Note that even though the goal of this paper is the C programming language, nevertheless AspectJ is well suited because of C and Java programming languages have many similar constructions and almost all AOP implementations are more or less based on AspectJ ideas.

```
// An aspect consisting of a named pointcut
// and an advice.
aspect Logging {
    // A named pointcut that matches a join
    // points set of method calls.
    pointcut move():
        call(void FigureElement.setXY(int,int)) ||
        call(void Point.setX(int)) ||
        call(void Point.setY(int));
    // An advice performing some actions before
    // execution of a matched by the given named
    // pointcut program point.
    before() : move() {
        System.out.println("about to move");
    }
}
```

Figure 1. Example of an AspectJ aspect for a graphical system logging

In using an AspectJ extension for Java a logging functionality for a graphical system can be extracted into an aspect showed in Fig. 1 [3]. In whole this means that before execution of each called method from the specified ones the given log message will be printed to a screen. The AspectJ weaver deals with Java program bytecode and after its work such the object code is obtained. This weaver is implemented as a part of a special compiler. The given example shows that an AOP implementation really strongly depends on a programming language and a program build process. Indeed there are more than 20 different AOP implementations just for the Java programming language. So the main goal, to separate cross-cutting concerns from the core ones for a given programming language, can be reached in the different ways.

The rest of the paper is structured as follows. Section II considers features of the C programming language and a typical build process of programs written in the given language. On the basis of these features and demands of the Linux driver verification project (it's considered there) requirements to an AOP implementation for the C programming language are collected. Section III describes related work and shows how different approaches meet the requirements pointed out in Section II. A suggested approach of an AOP implementation for the C programming language is introduced in Section IV. Section V estimates an application of the suggested approach. Section VI summarizes the work done and considers future work directions.

## II. REQUIREMENTS TO AN AOP IMPLEMENTATION FOR THE C PROGRAMMING LANGUAGE

Let us consider a typical workflow in building of a common C program and estimate how AOP conceptions may be related with different C constructions. It's worth while noticing that during the given consideration we won't restrict a C AOP implementation to represent just AOP constructions similar to the AspectJ ones as it's done by the most of AOP implementations. On the contrary we will try to describe an AOP implementation specific for the C programming language. Generally speaking it's assumed that such the implementation won't have any fundamental limitations for C cross-cutting concerns separation.

Fig. 2 illustrates 3 stages of a common C program build process, *preprocessing*, *compilation* and *linking*. Note that rectangles having dash line borders represent third-party components used by a program considered.

### A. Preprocessing

At the first stage a preprocessor in depend on passed preprocessor options includes necessary header files (both program's *h1.h*, *h2.h*, *h3.h*, ... and libraries' *lib1.h*, *lib2.h*, *lib3.h*, ...) into a program source code files *a1.c*, *a2.c*, *a3.c*, ... and expand macros there. More exactly it's the two main actions performed by a preprocessor but the rest ones aren't touched in the given paper. Both header files including and macro expansion may be related with AOP conceptions in the following way. Each file included to a given program source code file and even that source code file itself can be treated as a corresponding join point. Therefore the including process can

be modified by adding needed instructions before, after or instead of a given file. For instance, this helps to add some auxiliary preprocessor directives, function prototypes and so on. Macro expansion also can be altered in the similar way. So instead of (or before, or after) a substituted code we may put our own code that may deal with macro arguments as well as perform some required actions. Preprocessing is the essential C feature because of there is just few programming languages through all that supports it.

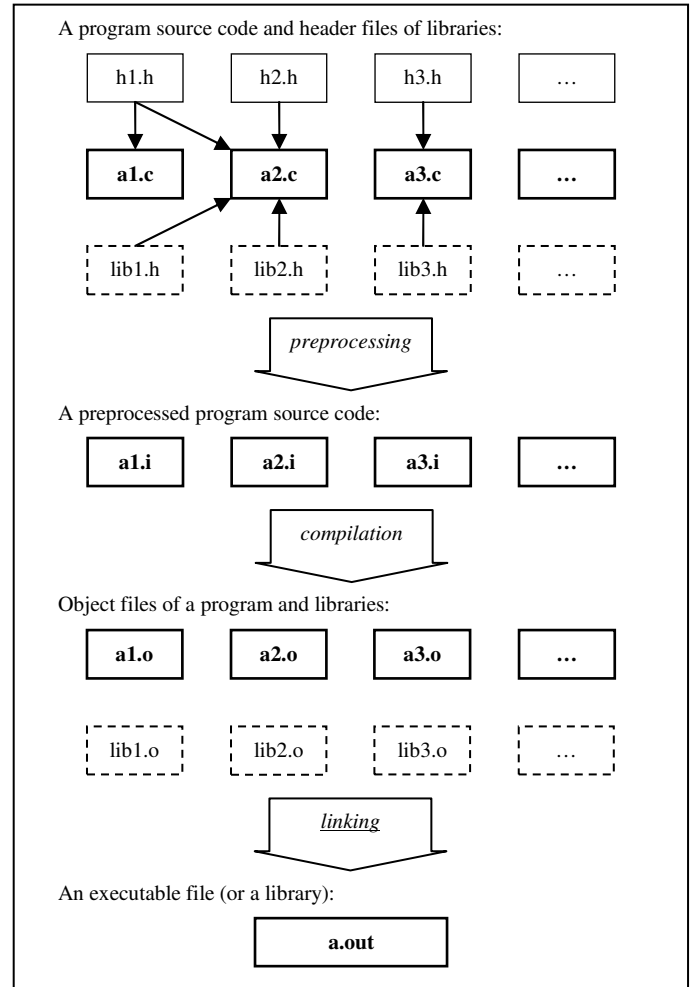


Figure 2. Common C program build process

### B. Compilation

Then at the second stage a compiler parses a preprocessed program source code files *a1.i*, *a2.i*, *a3.i*, ... and produces corresponding object files *a1.o*, *a2.o*, *a3.o*, ... Compilation may be affected by some build options. Traditionally AOP conceptions are developed for constructions of the given stage. For instance, AOP conceptions are related with function/method definitions and calls, type/class declarations and variable and field manipulations. Here is indeed the large area for AOP to be involved. As for the given work it's suggested that there should be implemented at least support for such join points as a function definition and call, a type declaration, a local and global variable, field and function

parameter set and get. The most of current AOP implementations support just the given or even a smaller set of join points (see Section III). Moreover C is a programming language having pointers and a lot of operations with them. The most popular operations like a pointer dereference and vice versa a taking of a variable/field/parameter address and a taking of an array element should be supported as corresponding join points. Also it's required that for each mentioned join point some actions written as advice bodies can be performed before, instead of and after a corresponding program construction execution. Of course it isn't a complete list of different program join points, e.g. loop and specific condition statements as well as a lot of different expressions weren't described. But indeed they also may be taken into account sooner or later.

Both the first and the second stages weaving should produce a correct source code or/and a corresponding compiler internal representation. For example, advice bodies should be substituted and be compliable as well as a given program source code. Also for these stages a considered construction scope (either some file or some function) plays a significant role because of some action like a function call may be performed either in one file or in another one, in one function or in another one. It is important to notice that among advice body instructions there may be some specific AOP instructions. In the given paper they are referred to as **body patterns**. For instance, there may be such body patterns as:

- a matched construction name and type (for a function call and definition, for a variable declaration, etc.);
- matched construction argument names and types (for a function call and definition);
- a matched construction size (for entities having an array type or strings);
- and even a matched construction itself (e.g. to have ability to call a matched function from inside a corresponding advice body).

### C. Linking

Linking performed at the third stage by a linker with corresponding build options assembles given program object files *a1.o*, *a2.o*, *a3.o*, ... together with libraries object files *lib1.o*, *lib2.o*, *lib3.o*, .... After all an executable file or a library *a1.out* is obtained. It's worth while mentioning that C program object files to be linked shouldn't contain the same defined symbols such as function definitions having the same names. So if some shared functions and global variables are required to separate cross-cutting concerns they should be contained just in one object file. For instance, this may help to use different counters or flags, i.e. to save a shared context or state, and to efficiently execute the same code by means of special auxiliary functions (one can see an example in Section V). Interaction of AOP directly with object files and a running program is beyond of the given paper.

So the common C program build process, the most of key C constructions and their influence on an AOP implementation were considered. But the goal of the given paper isn't to

introduce some AOP implementation for the C programming language but is to suppose the one that can be used for real programs. To the author's knowledge unfortunately the most of C AOP implementations are used just for artificial simple examples and isn't widely used in practice (there is some discussion about the given issue in Section III). So the AOP implementation concerned at this paper was strongly affected by the Linux driver verification (LDV) project [4][5]. The goal of that project is to provide an industrial quality toolset that allows to use different static code analysis tools to verify whether drivers satisfy correctness rules or not. The appropriate way to formalize these correctness rules in the manner being independent on a static verifier used and than to instrument a driver source code to be checked is to use AOP. Therefore this constrains some extra circumstances on a C AOP implementation:

- Support of the C programming language with all GNU extensions as an input language (it's a standard language for drivers writing) as well as all support of standard and GNU build options.
- Offering of a well set of AOP constructions corresponding to the C programming language. This is required since correctness rules refer to different C constructions used in different contexts. But nevertheless aspects development should be rather easy.
- An output should be also a correct program in C equivalent to the original one except it may be extended with corresponding cross-cutting concerns. This is required by the following application of static code analysis tools.
- An AOP implementation should be quite easy maintained and extended with new features. This comes because of new correctness rules are constantly appearing, so an extra AOP constructions support is required.

Note that nevertheless the most of these requirements are suitable for any program written in the C programming language (may be with allowance that this is done on the Linux platform). So a supposed C AOP implementation appears to be used both in the LDV project and in developing of a rather random C program. Moreover the requirement for an output to be a C program is useful for an AOP implementation debugging, because of by means of this output one can easily observe how a given AOP implementation behaves.

## III. RELATED WORK

AOP for the C programming language that is the goal of the given paper is considerably less developed in comparison with the one for Java. At present the most interesting C AOP implementation is ACC (AspeCt-oriented C) [6]. Fig. 3 shows that its superset for C likes the one for Java made in AspectJ [7]. That aspect means that after function *foo2* is called its result will be printed to a screen. ACC weaving differs from the one of AspectJ. For a given preprocessed C file ACC produces a corresponding C file extended with cross-cutting concerns. Despite of ACC supports a rather large set of AOP

constructions it cannot deal with preprocessor ones since it takes already preprocessed source code. Also it is intended just for one file processing and there isn't ability to specify some shared variables and auxiliary functions. ACC has its own closed C parser that fails to process some GNU extensions. Maintenance of ACC by its developers isn't active and due to its core component is closed it isn't so easy to deal with it.

```

// An advice printing a message after a
// given function call is performed.
after (int res): call(int foo2(int)) &&
result(res) {
    printf(" after call foo2, return %d\n", res);
}

```

Figure 3. Example of an ACC aspect

InterAspect is a more recent AOP implementation intended for the C programming language [8]. It was developed almost at that time when the given work was done. This tool is interesting because of it's based on GCC plugins [9], so it is most likely to support all GNU extensions. Unfortunately the InterAspect tool after all produces an object code (in fact this is done by GCC itself) like AspectJ so it cannot be directly used for static verification. At present the given tool supports rather limited number of AOP constructions and preprocessor constructions aren't supported as well as state variables and auxiliary functions. Instead of a C superset it provides a special C AOP library allowing to write aspects like an usual C program. But as one can see in Fig. 4 it seems to be even a more complex task to write such an aspect. In fact there only a joint point for *malloc* function call is defined. The tool was actively developed recently. However its development was stopped at the end of 2010. Nevertheless its progress should be tracked and correlated with the suggested approach.

```

static void instrument_malloc_calls() {
    /* Construct a pointcut that matches calls
to: void *malloc(unsigned int). */
    struct aop_pointcut *pc =
aop_match_function_call();
    aop_filter_call_pc_by_name(pc, "malloc");
    aop_filter_call_pc_by_param_type(pc, 0,
aop_t_all_unsigned());
    aop_filter_call_pc_by_return_type(pc,
aop_t_all_pointer());
    /* Visit every statement in the pointcut. */
    aop_join_on(pc, malloc_callback, NULL);
}

```

Figure 4. Example of a part of an InterAspect aspect

Another good approach is SLIC (Specification Language for Interface Checking (of C)) [10]. To the author's knowledge it's the only C AOP implementation that is widely used in practice. However it has just one field of application, it's used during a process of static verification of Microsoft Windows operation system drivers. SLIC allows to use state variables and has a simple syntax for aspect writing. A SLIC specification is indeed some kind of an aspect. The example of a SLIC specification is demonstrated in Fig. 5. This artificial specification states that it is an error to have more than four zeroes in a queue. A SLIC preprocessor weaves driver source

code with a specification and after all produces equivalent C program to be checked by means of a static verifier. A shortcoming of the given approach is that there just few join points are implemented (in fact just a function call and definition). Also the given project is completely closed.

```

state { int zero_cnt = 0; }
put.entry {
    if ($1 == 0) {
        if (zero_cnt == 4)
            abort "Queue has 4 zeroes!";
        else
            zero_cnt = zero_cnt + 1;
    }
}
get.exit {
    if ($return == 0)
        zero_cnt = zero_cnt - 1;
}

```

Figure 5. Example of a SLIC specification

A lot of other AOP implementations for C like C4, Aspicere2, Xweaver project, WeaveC and so on posses a less number of useful features than the ones described above, so they aren't considered in this paper. Also AOP tools dealing with C++ even though they may be adapted in some way for the C programming language aren't introduced because of usually they produce output in C++ while C is required by static code analysis tools.

#### IV. OVERVIEW OF SUGGESTED C AOP IMPLEMENTATION ARCHITECTURE

A suggested approach tends to implement all the requirements described in Section II in the most complete way. So after thorough investigation it was decided to base it on the LLVM compiler infrastructure [11]. In turn this infrastructure is built on top of GCC, it has so-called LLVM GCC Front End binding GCC with LLVM tools. So the LLVM compiler infrastructure inherits a GCC parsing of both C constructions and GNU extensions and supports all GCC build options almost as InterAspect described above. The suggested C AOP implementation is built on top of a GCC parser itself. Because of GCC includes preprocessing the given C AOP implementation can deal with both preprocessing and compilation join points. Next the LLVM tools include its own linker and a C backend tool. The first allows to link several object files of the whole program together, so some set of source code files can be woven instead of an alone file. The C backend tool is used to produce a C source code file to be verified by a static code analysis tool. To write aspect files it was decided to use a superset of C like AspectJ, ACC and SLIC do. Section V contains an example of such an aspect that is used in practice. Below the overall architecture of the suggested C AOP implementation is considered in more details. It's shown how program source code files, libraries' header files and aspect file are used and modified to weave cross-cutting concerns with a program.

Different constructions matching and weaving are performed through 4 stages by means of LDV GCC Front End invocation on each stage. Then linking and a C source code file generation are done. First of all it's necessary to mention that

there are usually 2 aspect files. The first is intended for weaving with all program source code files. The second aspect file is required to define auxiliary function definitions and global variable declarations shared between all other source code files. The second aspect file is applied just to one program source code file of those forming a final executable file or a library. To make the further description more general \* is used instead of corresponding names. For instance a first aspect file is denoted as *\*.aspect*, and a second as *\*.aspect.common*.

#### A. Aspect preprocessing

At the first stage comments of both C and C++ styles are eliminated from both aspect files. So *\*.aspect.nc* and *\*.aspect.common.nc* (where *nc* means “no comment”) are obtained. Then at every stage such the modified aspect files are parsed by means of a special parser (that is later referred to as *aspect parser*) implemented as a patch for LLVM GCC Front End. In an aspect file parsing lexical, syntax and semantic correctness is checked. Advice bodies are looked through just to determine body patterns. In case of some error an exact place and an error type are reported. If a given aspect file is correct it’s translated into own internal representation used during matching and weaving later.

At the first stage required modifications are done for a program source code file processed, *\*.c*. Either before or after or instead of it some additional source code is inserted. This is done to process further these modifications as soon as possible, i.e. even by means of a preprocessor because of they may contain some preprocessor directives. By analogy with a preprocessor a file obtained after this stage is called *\*.c.p* (*p* means “preprocessed”) and the given stage is named *aspect preprocessing*. At the moment there isn’t weaving for included files but it can be implemented in the similar way.

#### B. Macro weaving

At the second stage during the standard preprocessing of a *\*.c.p* file performed by LLVM GCC Front End using corresponding build options (e.g. to find all included files) macro matching and weaving are performed. So this stage is referred to as *macro weaving*. When a corresponding to a given pointcut macro directive is matched a macro body is extended in a way required by an advice. After all there is a *\*.c.p.mw* (*mw* means “macro woven”) file that is the both aspect preprocessed and preprocessed one.

#### C. Advice weaving

The third and the fourth stages correspond to the compilation phase. Here is important to notice that we don’t restrict an advice body source code with C constructions usage and we don’t parse it by ourselves. Instead, advice bodies are substituted to a given source code file as unique auxiliary function bodies on advice pointcut matching. And then the LLVM GCC Front End powerful parser processes them. So at the third stage auxiliary functions required to implement advice body actions are created in depend on join points matching and advice requirements. Also to perform parsing of type declaration extensions as well as to allow using of given extensions in auxiliary functions type declarations weaving is done at the third stage. At this stage the LLVM GCC Front End

C parser deals with a preprocessed file *\*.c.p.mw* and produces step by step its intermediate representation in the form of the GCC internal representation, called later as a *parsing tree*. Also parsed entities (in fact, type declarations and function bodies) are looked through to find matches with pointcuts defined in a given aspect file. It’s kept where matched entities are placed (to insert either auxiliary function prototypes or to extend corresponding type declarations later), what exact types and names are matched to replace body patterns used in corresponding advice bodies. After all required type declaration extensions as well as auxiliary function definitions with substituted body patterns and their prototypes are directly inserted into corresponding places of an initial source code file *\*.c.p.mw* and a *\*.c.p.mw.aw* (*aw* means “advice woven”) file is obtained. The stage is called *advice weaving*.

#### D. Compilation

After that at the fourth final stage the inserted source code is checked for correctness and translated into a parsing tree as well as an initial source code. Also at the third stage matching and weaving are performed in parsing. Here function definitions and function body expressions are modified directly at the level of the parsing tree and some relations with auxiliary functions are established if it’s necessary. After the parsing is completed LLVM GCC Front End behaves in its standard mode and obtains an object file as well as a compiler does.

All four stages described above are summarized in Table 1. The table shows how input data is modified and used and what output is obtained in depend on a given stage.

TABLE I. DATAFLOW OF MATCHING AND WEAVING STAGES

Stage	<i>*.aspect</i>	<i>*.c</i>	Build options
<i>Aspect preprocessing</i>	Comments elimination ( <i>*.aspect.nc</i> ) and parsing	Include join point weaving ( <i>*.c.p</i> )	Aren’t used
<i>Macro weaving</i>	Parsing	Macro weaving and preprocessing ( <i>*.c.p.mw</i> )	Preprocessor options are used
<i>Advice weaving</i>	Parsing	Auxiliary functions and declarations direct including ( <i>*.c.p.mw.aw</i> )	Compiler options are used
<i>Compilation</i>	Parsing	Function definitions and bodies weaving, compilation	Compiler options are used

#### E. Linking and C source code file generation

Further required object files are linked together by means of the LLVM linker tool. As it was already mentioned for a resultant file just one object file woven with both aspect files is taken. For an assembled object file the LLVM C backend tool produces a C source code file that can be processed by a static verifier. Although the last action is optional. For example, instead of this there is ability to produce an executable file for a given program that is intended for some architecture supported by the LLVM compiler infrastructure.

### V. APPLICATION OF SUGGESTED C AOP IMPLEMENTATION

The suggested AOP implementation for the C programming language is already included into a LDV project toolset. It’s

used to formalize few correctness rules and in driver source code instrumentation intended for a further verification by means of static code analysis tools.

Fig. 6 shows an example of aspect files used in verification of the “Locking a mutex twice or unlocking without prior locking” correctness rule. Note that these aspect files are simplified in comparison with the actually used ones since some extra lock functions aren’t presented. Syntax is most likely to be rather intuitively clear. It’s worth while noticing that there are 2 join points as for macro *mutex\_lock* and for function *mutex\_lock*. This is required because of Linux kernel can define either a macro or a function in depend on its configuration. Function *mutex\_unlock* is always declared as extern, so it doesn’t require instrumentation since it can be explicitly defined. Global variable *ldv\_mutex* is an example of shared state variables while *ldv\_mutex\_lock* is an auxiliary shared function. Function *ldv\_check\_final\_state* is executed at the end of checking to ensure that nothing is locked then.

```

model0032a-blast.aspect
before: file ("$.this") {
#include <linux/kernel.h>
#include <linux/mutex.h>
extern void ldv_mutex_lock(struct mutex *lock);
}
around: define(mutex_lock(lock)) {
ldv_mutex_lock(lock)
}
before: call(extern void mutex_lock(struct
mutex *)) {
ldv_mutex_lock($arg1);
}

model0032a-blast.aspect.common
after: file ("$.this") {
#include <linux/kernel.h>
#include <linux/mutex.h>
#include "engine-blast.h"
int ldv_mutex = 1;
void ldv_mutex_lock(struct mutex *lock) {
ldv_assert(ldv_mutex == 1);
ldv_mutex = 2;
}
void mutex_unlock(struct mutex *lock) {
ldv_assert(ldv_mutex == 2);
ldv_mutex = 1;
}
void ldv_check_final_state(void) {
ldv_assert(ldv_mutex == 1);
}
}

```

Figure 6. Example of aspect files of the supposed C AOP implementation

To estimate quality of the proposed C AOP implementation 2 experiments were performed. The first one used a specially prepared Linux kernel configuration and corresponding kernel function implementations (like *mutex\_unlock* showed in Fig. 6) while the second one used aspect files like presented in Fig. 6 and following instrumentation. Later the first experiment is called *plain* and the second one is called *aspect*. During experiments all drivers of Linux kernel 2.6.31.6 [12] that can be represented as kernel modules (there are 2160 such the drivers) were examined against the correctness rule about mutex lock/unlock described above with help of BLAST static

code analysis tool [13]. The most interesting results demonstrating verdict changes between plain and aspect approaches are shown in Table II (a first verdict belongs to the plain experiment, and the second one belongs to the aspect one). *Safe* verdict means that a given driver satisfied the given correctness rule, *unsafe* is the reverse one, *unknown* verdict means that a static verifier used failed to check a given driver (e.g. because of time or memory shortage or due to some parsing error).

TABLE II. COMPARISON OF THE SUGGESTED C AOP IMPLEMENTATION WITH ANOTHER APPROACH

Safe → Unsafe	Safe → Unknown	Unsafe → Unknown	Unknown → Safe	Unknown → Unsafe
4	95	18	82	3

As one can see from Table II the supposed C AOP implementation behaves rather well because of the number of “bad” transitions (i.e. from safe/unsafe to unknown) almost equals to the number of “good” transitions. There are 95 + 18 = 113 “bad” transitions and 82 + 3 = 85 “good” ones. Their difference is just 28, that is less then 1.3% of the total number of kernel modules.

In fact it requires more memory for a generated file verification to be performed in the aspect experiment in comparison with the plain one. So, 62 modules were not checked because of memory shortage. Also in the aspect experiment some produced by LLVM C backend C constructions are rather complex for the static verifier used (31 modules were not checked due to the given reason). Although the plain experiment showed that even more drivers confuse a BLAST C parser because of complex constructions coming from initial driver source code as is. There are 68 such modules. The rest transitions from/to unknown verdict are concerned with either some bugs in the supposed C AOP implementation (20 modules for the aspect experiment) or time/memory shortage in the plain experiment (17 modules). Unfortunately, all additionally found unsafes (7 modules for which safe or unknown verdict was exchanged with unsafe one) are false positives because of either generated C file shortcomings (like generation of big unsigned integer numbers instead of negative ones that is demonstrated later) or incomplete correctness rule implementation and some static verifier lacks.

But nevertheless the most significant shortcoming of the supposed C AOP implementation consists in a generated code itself. Fig. 7 illustrates an example of how a driver source code is modified after the given implementation invocation. As it was already mentioned sometimes this prevent a static verifier from check performing due to complex constructions generated. As Fig. 7 shows there is a lot of variables having prefix *blast\_must*. This is a special workaround made as a corresponding LLVM C backend patch. It is required to designate so-called *must-aliases*, that is the aliases that alias only one known memory location (all artificial temporary variables are must-aliases). The suggested approach application leads to more memory requirement for a testing to be executed. Such the generated source code scares users trying to see on it, for example, in analyzing unsafes or in debugging the given C AOP implementation. In fact the LLVM compiler

infrastructure used is responsible for this shortcoming. First of all it deals with a GCC internal representation called GIMPLE that already rather differs from a source code pure representation. Next it is intended for machine independent source code generation. So one can see large positive numbers instead of small negative ones in Fig. 7.

```

drivers/pci/hotplug/fakephp.c (preprocessed)
  if (strict_strtoul(buf, 0, &val) < 0)
    return -22;
  if (val)
    pci_rescan_bus(slot->dev->bus);

fakephp.ko.linked.cbe.c
  blast_must_tmp__85 = *(&llvm_cbe_buf_addr);
  blast_must_tmp__86 =
strict_strtoul(blast_must_tmp__85, 0u,
(&llvm_cbe_val));
  if (((signed int)blast_must_tmp__86) <
((signed int)0u))
    goto llvm_cbe_bb;
  else
    goto llvm_cbe_bb1;
  llvm_cbe_bb:
    *(&llvm_cbe_tmp__73) =
18446744073709551594u11;
    goto llvm_cbe_bb5;
  llvm_cbe_bb1:
    blast_must_tmp__87 = *(&llvm_cbe_val);
    blast_must_tmp__88 = *(&llvm_cbe_slot);
    blast_must_tmp__89 = *(&blast_must_tmp__88-
>field1));
    if ((blast_must_tmp__87 != 0u11))
      goto llvm_cbe_bb2;
    else
      goto llvm_cbe_bb3;
  llvm_cbe_bb2:
    blast_must_tmp__90 = *(&blast_must_tmp__89-
>field1));
    blast_must_tmp__91 =
pci_rescan_bus(blast_must_tmp__90);
  llvm_cbe_bb3:

```

Figure 7. Comparision of a driver source code with the generated one

Another big shortcoming is connected with the fact that LLVM GCC Front End is based on the rather old GCC compiler (of 4.2.1 version, nowadays 4.5.2 is a stable release) while the modern Linux kernel drivers already posses such new constructions that aren't processed with it. So different workarounds are required to overcome this.

After all let us imagine how different approaches introduced in Section III could meet aspect files presented in Fig. 6, driver source code instrumentation, following static analysis and obtained verification results examination. First of all none of them supports the join point concerned with the preprocessor construction `define(mutex_lock(lock))`. Then, step by step, ACC fails to parse driver source code because of unsupported fresh GNU extensions to the C programming language and that tool cannot be adjusted because of it uses a closed parser. InterAspect deals with GIMPLE representation of source code and, if we had some C backend tool for GCC, InterAspect would produce instrumented source code too dissimilar to the original one almost as well as LLVM C backend. Both ACC and

InterAspect doesn't support state variables and functions like `ldv_mutex` and `ldv_mutex_lock` correspondingly. Most likely that we could verify the given model by means of SLIC, except the preprocessor issue, but in fact this is one of the simplest model from the LDV project. Other models require more complex join points and advice bodies, so what can we do if SLIC supports just function calls and definitions and it is the closed project.

## VI. CONCLUSION

This paper describes an approach of how to implement aspect-oriented programming in the way specific for the C programming language. It considers features and shortcomings of current implementations. After all a new implementation that tends to cover all major features of the C programming language as well as to take into account those features that come from the C programs build process is considered. It's shown how the given C AOP implementation behaves to reach the required intention.

For the supposed C AOP implementation its real application for the Linux driver verification process is demonstrated. An example of real aspect files implementing a correctness rule associated with the mutex lock/unlock problem is given. Also the supposed approach is compared with another one that doesn't use AOP. It's shown that the given C AOP implementation is rather good except a generated source code is too complex for further analysis and it's quite unlike the original one. Mental comparison with another AOP approaches, such as ACC, InterAspect and SLIC, is done. Finally it becomes clear that the given approaches can not meet all requirements imposed on the suggested C AOP implementation by a number of reasons.

The current development of the supposed approach of the AOP implementation for the C programming language tends to overcome the restrictions specified above. To keep all advantages of the supposed approach as well as to eliminate the given shortcomings it was decided to develop our own C backend tool intended directly for GCC itself. It's assumed that it'll be built on top of stable GCC "from svn" that is it'll parse all modern constructions and GNU language extensions. Also the given C backend tool should work at the low-level GCC internal representation even before GIMPLE. Thus far a produced source code will most likely to be very similar to the original one. We believe that this will allow to combine abilities of both the supposed C AOP implementation and powerful GCC compiler to process C source code and to use AOP.

One can obtain the current AOP implementation for the C programming language from a LDV development site [14]. There it can be found as a part of *rule-instrumentor*. It's planed that an updated C AOP implementation will also be there soon.

## REFERENCES

- [1] Definitions of key AOP concepts. [http://www.aosd.net/wiki/index.php?title=Main\\_Page](http://www.aosd.net/wiki/index.php?title=Main_Page)
- [2] AspectJ: an aspect-oriented extension to the Java programming language. <http://www.eclipse.org/aspectj/>
- [3] An AspectJ example.



- <http://eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html>
- [4] A. Khoroshilov, V. Mutilin, V. Shcherbina, O. Strikov, S. Vinogradov, and V. Zakharov, "How to cook an automated system for Linux driver verification," 2nd Spring Young Researchers' Colloquium on Software Engineering, vol. 2, pp. 10-14, 2008.
  - [5] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov, "Establishing Linux driver verification process," Perspectives of Systems Informatics, vol. 5947 of Lecture Notes in Computer Science, pp. 165-176, 2010.
  - [6] M. Gong, C. Zhang, and H.-A. Jacobsen, "AspeCt-oriented C," Technology Showcase, CASCON 2007, Markham, Ontario, 2007.
  - [7] W. Gong and H.-A. Jacobsen, "AspeCt-oriented C Language Specification Version 0.8," University of Toronto, 2008.
  - [8] J. Seyster, K. Dixit, X. Huang, R. Grosu, K. Havelund, S. A. Smolka, S. D. Stoller, and E. Zadok, "Aspect-Oriented Instrumentation with GCC," Proceedings of the First International Conference on Runtime Verification, pp. 405-420, 2010.
  - [9] GCC plugins. <http://gcc.gnu.org/wiki/plugins>
  - [10] T. Ball and S.K. Rajamani, "SLIC: a Specification Language for Interface Checking (of C)," Technical Report MSR-TR-2001-21, Microsoft Research, 2002.
  - [11] The LLVM Compiler Infrastructure. <http://llvm.org/>
  - [12] Linux kernel 2.6.31.6. <http://www.kernel.org/>
  - [13] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker Blast: Applications to software engineering," Int. J. Softw. Tools Technol. Transf. 9(5), pp. 505-525, 2007.
  - [14] The LDV project. <http://forge.ispras.ru/projects/ldv>

# Component-Based Software Engineering and Runtime Type Definition

A. R. Shakurov  
Business Informatics Department,  
Higher School of Economics,  
Moscow, Russia  
amir-shak@yandex.ru

*Abstract*—The component-based approach to software engineering, its current implementations and their limitations are discussed. A new extended architecture for such systems is presented. Its main architectural concepts and principles are considered.

*Index Terms*—Runtime environment, software architecture, software engineering, software reusability

## I. INTRODUCTION

SOFTWARE ENGINEERS have always pinned their hopes on the idea of **reusable code** [8]. In their urge towards eliminating code duplication, simplifying code maintenance, making it less error-prone and streamlining the development process programmers have gone a long way from a completely unstructured code through procedures and program libraries to object-oriented technologies and application frameworks. Taking the code reuse idea one step further, the component-based software engineering [9], [14] is a very promising approach to software development.

The term “**component instance**” usually refers to a program entity holding data and offering some functionality that are hidden by a well-defined interface (cf. [10], [11]). The concept of interface, however, varies from one technology to another: a number of “properties” (attributes, member variables etc.) [13], a number of member functions/methods (as in most object-oriented programming languages) or even an entity whose nature may be left out of scope of the component technology itself [1]. The process of combining components into a working system is considered to be relatively simple (e.g. [2]). Nevertheless, this process is implemented differently in various technologies.

In order to propose an optimal way for organizing component interaction, that would introduce a good balance between flexibility and ease of use, let us concisely consider main aspects of component-based models and technologies and their limitations (detailed comparative analysis is beyond the scope of this work, see in [12] and [14]).

## II. LIMITATIONS OF COMPONENT-BASED TECHNOLOGIES

Despite the many advantages of the component approach its currently existing implementations have a number of substantial limitations. The most difficult goal to achieve here is probably to find a way of designing components that will provide the necessary functionality without exceeding it. The necessary functionality is determined by requirements, and these are bound to change with the lapse of time. There're three options to consider. First of all, it is possible to introduce software with somewhat wider capabilities, so that it would still be adequate when requirements change. This approach, however, demands remarkable architectural design skills and foresight and has the risk of bloating the program under development making it unsuitable for system with limited memory resources. Second of all, one can adapt the software with the course of time. This would result in the most appropriate as well as the most expensive software. The third option is to introduce a component framework that would allow in-place modification of component's functionality without stepping over the bounds of the component model.

Let us consider an example. ZigBee specification [15] offers a suite of high level communication protocols for low-rate, low-cost, low-power-consumption wireless personal-area networks. It is implemented, for instance, by communicational parts of microelectromechanical systems in wireless sensor networks. At the same time, the specification is subject to frequent changes. This makes manufacturers renew and release sensor firmware, which is a difficult process due to the lack of high-level development tools for such systems.

ZigBee specification introduces network and application layers (in addition to the PHY and MAC layers defined by the IEEE standard 802.15.4 [5]) to the protocol stack. The topmost application layer (which is subject to frequent changes) is comprised of a number of components: ZigBee device objects, their management procedures, application objects. Requirements to these components are not changed

at the same time (application objects, for example, are provided by the manufacturer and thus aren't managed by the ZigBee alliance). Nevertheless, every new release of the specification implies a new release of the whole firmware. It would be more cost-efficient to modify only those components specification for that have changed and make the system reconfigure to make use of their instances without rewriting a binary firmware image to devices' memory.

To add, remove or modify certain functionality of a component's instance means to introduce a new component, because it is the component that defines functionality of its instances. That is, we're essentially dealing with the task of defining a **new type of data**. This can be done either by deducing new type from existing one or creating it from scratch using a number of predefined low-level components. We believe it's this operation that must be available at runtime in order to ensure software flexibility.

This problem can be tackled in a number of bypass ways such as source code, bytecode or binary code generation/transformation, runtime compiler calls or taking advantage of programming language's ability to modify its low-level runtime data structures (Java Reflection being a graphic example). These, however, aren't always an option. Reflection mechanisms are primarily meant to be used to build IDEs, source code analysis tools and GUI designer applications. Using them indiscriminately to build any garden-variety software lays exceedingly high claim to developer and is therefore error-prone. Furthermore, many programming languages don't support reflection at all. And embedded systems often lack compilers and code modification frameworks. Therefore an alternative system capable of building new types from user-configured instances without stepping over the bounds of its model is needed.

To conclude the section, let us look at another simple example. We're designing a GUI application and we want to change a button's label (the button has already its functionality attached to the application). The change is supposed to be permanent: the button won't be renamed at runtime. Although obvious, the necessary procedure is implemented in various frameworks with a distressing flaw: the variable property is left variable for the lifetime of the component's instance regardless of developer's intentions. In other words, the fact that the button's text isn't meant to be mutable at runtime (should this be the designer's intention) cannot be expressed by a developer.

The designtime work with an instance of a component and the runtime work with it are basically two different contexts of its use. However, these cannot be fully separated by existing frameworks (see [13] e.g.), because one has to start (execute) a component (i.e. instantiate it) in order to either configure it or take advantage of its functionality. This is the source of the troubles one encounters when

adjusting a component's instance.

To summarize, a new component system capable of introducing new data types by either modifying existing ones or creating them from scratch using a number of predefined low-level components is needed in order to create flexible (adaptable to changing requirements and contexts of use) software for systems with limited resources.

### III. CORE PRINCIPLES

To achieve the goal of designing such a system, we have analyzed advantages and limitations of existing object-oriented programming languages and component technologies. This has allowed us to infer the core principles of the suggested component model described in the next section.

The model is based on a general object-oriented idea, and is extended by other technologies' traits when necessary since the need to define data types at runtime makes designtime and runtime essentially indistinguishable. Omitting a detailed consideration of specific technologies here (see [12] and [14] for such review), we are going to summarize the core principles that underlie the model in question.

The two primary characteristics of any development and execution environment are the principles and mechanisms of data organization and control flow management. Looking for those characteristics in the object-oriented paradigm, one will find the **hierarchical data organization** principle and **the concept of method** as the means of control flow management. And while the first characteristic gives us a well-balanced solution for managing ever growing complexity of software systems, we find the second one to be too complex and cumbersome. We believe that the very concept of object method (see [3] e.g.) isn't suitable for adopting it in a component model, because of its overwhelming versatility: a method can have variable number of parameters and (in certain languages) return values, or it can have none of those; arguments can be passed by either value or reference; methods can be overloaded and overridden (in which case a complicated set of resolution rules takes the stage). The list can be continued. The concept of method, therefore, doesn't provide intended ease of use (though it does handle software complexity well).

In contrast, the concept of **property** introduced in some frameworks (C#, JavaBeans) is more suitable to our needs. The property-based interaction model is simple and formal because of the limited number of aspects describing the "property" concept: its type and applicable operations (usually reading and writing). However, to become a perfect rival (to efficiently implement callback routine, for example) the concept needs to be extended with the binding operation (described in following section).

To summarize, we have adopted the principle of hierarchical data organization and the property concept as the means of organizing execution flow to create a component model with a runtime data type definition capability. We will now proceed to describe the model.

#### IV. THE MODEL

The model we're going to discuss has its prototype implemented in the Java programming language. Therefore let us start with describing the model from the user's point of view.

While working with the application, a user interacts with three categories of objects, viz. components, components' instances and containers (which are used as both runtime and new type definition environments).

##### A. Components and instances

An instance of a component is an aggregate of data and behavior hidden behind an **interface**, the latter being the only way to interact with this kind of entity. We shall refer to the hidden part of an instance as its **implementation**, as opposed to the interface.

Like objects in object-oriented paradigm have classes, instances have components that describe the way these kinds of instances are created and function. Every instance has a single component associated with it and this association is immutable during the lifetime of an instance. Every component can be instantiated without providing any additional information. This means that primitive types of data (numerical, boolean, string etc.) aren't actually components. These are usually called value-types and to instantiate them one has to provide at least a value of the instance to create.

In addition to contextless instantiation, components allow instantiation in a certain context (e.g. as part of a composite instance, see below).

##### Interface

Let us focus on an interface of a component's instance. It is comprised of a number of properties each of which is characterized by:

- name (used to identify a property),
- value type,
- access permissions; any property may be accessible for reading, writing and binding.

Read/write operations need no explanation. **Binding**  $S$  property of a particular instance to  $D$  property of another instance ensures that whenever the value of  $S$  is changed the new value will be written to  $B$ . The binding operation, as it was mentioned earlier, is needed to efficiently implement callback routine and is similar to that of the Java Beans model, except that there's only one event type (property

value change event) whereas in Java Beans a property can have multiple event types associated with it.

##### B. Container

A **container** is an execution environment that allows the following operations to be performed in its context ("within it"):

- instantiate any components,
- change values of properties of instances created at the previous step,
- bind instances' properties to each other.

Apart from that, a container can be used to create new components. Its contents are considered to be a prototype of an implementation part of a future component. To create a new component, one has to complete this information with interface specification and connect these two parts together. Above that, a user is able to edit metadata of instances constituting future implementation. All this can be achieved with the following operations:

- restricting access to instances' properties,
- adding properties (with specified metadata: name, access permissions etc) to the interface part of the component,
- adding **sharing connections** between a property of an instance that is a part of the implementation and a property of the interface.

The "sharing connection" between two instances' properties makes those instances share memory cell (therefore changing the value of one property is immediately reflected on the value of the other property). This behavior requires a custom memory model, which we'll focus on later.

Once the user has provided all the necessary data to a container, he can create a new component that will correspond to the prototype currently in the container in the sense that all of its future instances shall have the same structure.

While a user is working with a container, the latter gathers all the necessary information about future component. It is to be noted that some of this information can be deduced from the runtime structure of instances the user have created. For example, bindings between instances' properties are analyzed only when a new type is being created. Other pieces of information, however, can't be stored within the runtime structure. For example, restricting access to a property of an instance won't actually modify that instance because that implies changing metadata (i.e. data type) of an existing instance and there's no sense in doing that. In other words, not every user action can be directly reflected on the runtime structure of instances, and it's container who makes the process of editing both data and metadata transparent to a user.

It wouldn't be possible to define new types at runtime, however, if it wasn't for a specially designed internal structure of components, which will be discussed in the following section.

### C. Internal structure

Let us focus on the internal architecture of the prototype application we've developed that provides the previously described functionality. As it was mentioned above, the application is implemented Java, but it can be easily rewritten in any other strongly-typed object-oriented programming language.

All the instances in the system implement common **Instance** interface that provides methods to access instance's type and properties. Similarly, all the types implement the **Type** interface that provides methods to instantiate the type and also extends the Instance interface. Therefore, any type (and any component which is a kind of type) is an instance whose data are metadata describing its future instances. There is also a **TypeType** type, which is a type of any type (including itself).

Let us consider the following scenario. A user defines a new component with no bindable properties. Then the whole event-listener infrastructure (that supports bindings' functionality) becomes redundant and should not be included in corresponding instances. This kind of deep context adjustment is crucial when dealing with runtime type definition and we pay great attention to it.

To implement the smart adjustment described, we've introduced indirect access to instances' properties. They're accessed via special **PropertyGetter**, **PropertySetter** and **PropertyBinder** objects. If there're no bindable properties in a component then its instances end up having their PropertyBinder object uninitialized. And if there's at least one bindable property, then the binder object will be created (but it still won't be granting access to unbindable properties, of course).

### Memory model

As it was mentioned above, the custom memory model is required in order to consistently handle binding and sharing connections between properties.

The memory model introduces traditional kinds of "memory cells" (viz. constants and variables) that store instances as their values. The value can be read and (in case of a variable) written. The memory cell can also have no value (it is said to be null in this case). Finally, memory cells are strongly typed, which means that every cell knows its type and an attempt to store an instance of mismatching type in it produces an error.

There's also an unconventional kind of variable – listenable variables. As the name suggests, listenable variables (in addition to having all the features of regular variables) allow special objects (called listeners) to

subscribe to the variable value change event.

### Data types

We've already mentioned that the system supports primitive value-types which are a kind of instance types. Another kind of instance types is components, but there're two different kinds of them. The first one is **compiled components**. These are components whose implementation is not analyzed by the system in any way. This permits the system to handle various components implemented using a third-party means (e.g. java bean components).

The second kind of components is **composite components**. These are components implemented by means of the system itself. The implementation part of a composite component is a structure of other components.

Since it is a composite component that is built whenever user defines a new type, this kind of components is of greatest interest.

### Composite components

We are going to focus on the structure of metadata stored in composite components. Since these metadata determine the structure of data in corresponding instances, we will discuss that structure incidentally.

Composite type (like any other type) describes interface and implementation parts of its instances (Figure 1). These parts are interconnected via mechanism described below.

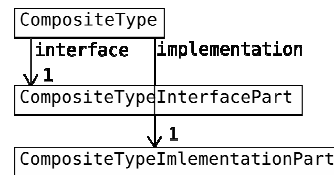


Figure 1. Composite type.

An interface of an instance is a set of its properties, so the metadata stored in the interface part of a component are a set of property descriptors (Figure 2) each of which specifies:

- property value type,
- access permissions,
- default value (optional).

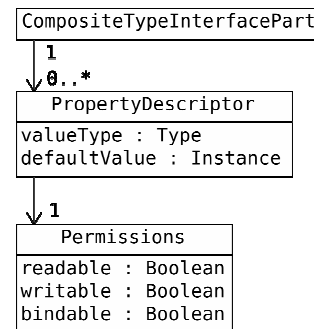


Figure 2. Interface part of a composite type.

Finally, implementation part of a composite type is metadata that describe a structure of instances with interconnected properties that will result from instantiation the type. These metadata are represented in the following way (Figure 3).

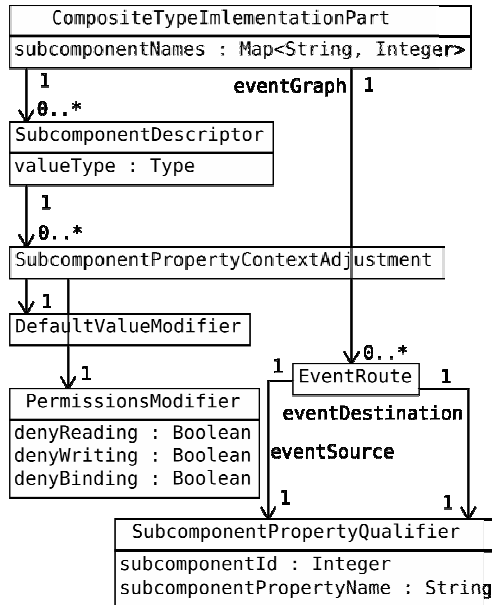


Figure 3. Implementation part of a composite type.

To every connection between two instances that constitute an implementation of the future component corresponds an object of the EventRoute class that holds two objects of the SubcomponentPropertyQualifier class (for the beginning and the ending of the connection). These objects simply specify a source and a destination of a property change event: they hold respective components' IDs and their property names.

For every instance that is a part of an implementation of the future component there's a descriptor, an object of class SubcomponentDescriptor. This object specifies:

- instance type,
- for every property of the instance, context adjustment of that property to its use as a part of another instance's implementation. This adjustment defines modified default value (if any) of the property (the DefaultValueModifier class) as well as restricts access to it (PermissionsModifier).

A default value of a property can be modified in a number of different ways.

- "Void" modification, the value is left intact.
- Explicitly specified value (the object of the DefaultValueModifier class holds this value).
- Value is another instance that's present in the same context (implementation of the same instance). In this case the modifier object holds the name of that instance.

- The property is shared. In this case not the value of the property, but the property itself is changed: instead of creating a new memory cell for storing the value, an existing (provided from elsewhere) cell is used by the instance.

The last option is used to provide an instance (via its instantiation context) with references to its parent instance's properties, thus creating sharing connections between these properties. These connections glue interface and implementation parts of a composite instance together.

### Composite component instantiation

To demonstrate described structure at work, let us go through the process of composite type instantiation. The following algorithm implements the process.

- 1) Memory cells for storing values of properties are established (one for every property descriptor). If there's an instantiation context, a reference for an existing cell is used. Otherwise a new memory cell is created, its kind (constant, variable or listenable variable) being determined by respective access permissions, and is initialized with either default value (if any) or a new instance of the corresponding type.
- 2) Instances constituting the implementational part of the future component are created. This involves evaluating their instantiation contexts. Every property descriptor is merged with corresponding subcomponent property context adjustment object giving a new set of adjusted access permissions and default value.
- 3) Binding connections (i.e. event routes) are established.
- 4) Proxy objects for accessing new instance's properties are created (property getters, setters and binders). These objects receive references to relevant properties only (e.g. a setter object only holds references to writable properties) and if there aren't any, the object is not created at all.

After all the steps have taken place, a new object of class CompositeInstance is constructed. The constructor is provided with the type (an object of the CompositeType class) and proxy objects. This results in a new instance of the component.

The described internal structure of composite components ensures great flexibility. Since data types are defined by the (runtime) structure of regular (java) objects (as opposed to compiled binary or bytecode), it gives us an opportunity to easily manipulate that structure at runtime thus creating new types.

## V. RESTRICTIONS AND FUTURE WORK DIRECTIONS

Let us discuss certain limitations to the described

solution. First of all, flexibility comes at a price. The ability to reconfigure software results in overhead computational costs. This means that the proposed model should be adopted only when implementing systems that either have no severe restrictions on computational resources or do not require very high performance. Aforementioned microelectromechanical sensors offer a graphics example: while having limited memory capacity, they carry no considerable performance limitations (they usually stay idle for hours between sending a signal and going idle again). This is why the availability of remote reconfiguration means takes precedence over elevated performance here.

Second of all, while the property-based interaction is simple and quite flexible, it has its limitations, too. For example, implementing intricate algorithms this way is possible though burdensome, making a traditional imperative-scripting style far more suitable choice. In other words, the described solution should be adopted when there're a great number of objects (instances) with a relatively simple interaction. In addition, it's possible to incorporate complex logic via compiled components, though this still requires implementing it in a third-party language.

The described system being only a prototype, our first and foremost goal is to turn it into a complete, feature-rich production-quality platform. This requires both evolving the component model and improving development tools to make use of it. This also implies optimization to guarantee acceptable performance.

As for the practical applications, we're planning to make use of the platform in question to introduce software solutions for 2D (GUI development) and 3D (VRML [6], [7] implementation) design, reconfigurable wireless sensors firmware and possibly for some other purposes.

## VI. CONCLUSION

We have described the main ideas and the core principles of internal organization of the new component architecture with extended capabilities. The ease of manipulating both data and metadata structure of software is not to be underestimated. We believe that formalized, simple yet powerful component model with runtime data type definition capability will allow creating most configurable software that will be able to evolve and adapt to changing requirements easily.

## REFERENCES

- [1] Bruneton, E., Coupaye, T., Stefani, J.B., *The Fractal Component Model specification. Version 2.0-3*, The ObjectWeb Consortium, 2004.
- [2] Costa Seco, J., Silva, R., Piriquito, M., "ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration", *Computer Science and Information System*, ComSIS Consortium, Novi Sad, Serbia, 2008, pp. 63-86.
- [3] Gosling, J., Joy, B., Steele, G., *The Java™ Language Specification. 3<sup>rd</sup> ed.*, Addison Wesley, 2005.
- [4] Heineman, G.T., Councill W.T. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, 2001.
- [5] IEEE Std 802.15.4-2003 – Wireless Medium Access Control (MAC) and Physical layer (PHY) for Low-Rate Wireless Personal Area Networks (WPANs).
- [6] ISO/IEC 14772-1:1997 — Virtual Reality Modeling Language (VRML).
- [7] ISO/IEC 14772-2:2004 — Virtual Reality Modeling Language (VRML).
- [8] Krueger, C.W., "Software reuse", *ACM Comput. Surv. Vol. 2*, ACM, New York, 1992, pp. 131-183.
- [9] McIlroy, M.D., "Mass produced software components", *Naur P., Randell B., "Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968"*, Scientific Affairs Division, NATO, Brussels, 1969, pp. 138-155.
- [10] Object Management Group, *The Common Object Request Broker: Architecture and Specification. Version 3.1. Part 3 - Components*, OMG document formal/2008-01-08, 2008.
- [11] Redmond, F.E., *DCOM: Microsoft Distributed Component Object Model*, IDG Books Worldwide, Inc., Foster City, 1997.
- [12] Stiemerling, O., *Component-Based Tailorability*, Bonn University, Bonn, 2000.
- [13] Sun Microsystems Inc. *The JavaBeans™ API specification. Version 1.01-A*, Sun Microsystems Inc., 1997.
- [14] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, 2nd ed, Addison-Wesley Professional, Boston, 2002.
- [15] ZigBee Alliance, *ZigBee Specification*, ZigBee Document 053474r17, 2007.

# Educational tests in “Programming” academic subject development

Maksimenkova Olga  
National Research University Higher School of  
Economics  
Moscow, Russia  
e-mail: omaksimenkova@hse.ru

Vadim Podbelskiy  
National Research University Higher School of  
Economics  
Moscow, Russia  
e-mail: vpodbelskiy@hse.ru

**Abstract**—Educational tests are quite popular as a type of learning outcomes checking in public education and in commercial education nowadays. This work is devoted to test for students, who study programming in Universities. Educational tests in “Programming” academic subject development and statistical analysis principles are described and accumulated here.

**Keywords:** *software engineering education, educational tests, tests, programming academic subject*

## INTRODUCTION

Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering recommends such courses as “Computer Science”, “Programming basics”, “Programming” etc. to be taught to train an undergraduate in Software Engineering. Nowadays in Russia syllabuses of subjects, which are widely connected with programming (for example, “Programming” or “Information Technologies”, etc.), obligatory contain a part about one or another programming language. The most popular programming languages: C++, C# and Java are taught all over the Universities.

The problem of checking theoretical knowledge of syntax and semantic of a programming language and practical skills is significant and quite complicated. Well-prepared educational test can help in resolving all these problems by checking all of them. It should be said that educational testing is widely applied practically in specialists’ certification by such firms as Microsoft, Cisco, IC etc.

This work is devoted to common questions of educational tests statistical analysis and development for students, who study programming in Universities. The research which is described in this work is based on tests in C# programming language which are given to the students who study programming within the academic subject “Programming” which is contained in the discipline of the Software Engineering.

## ESTIMATING OF LEARNING OUTCOMES

Desire learning outcomes of a process of learning are formulated by the academic staff, preferably involving student representatives in the process, on the basis of input of internal and external stakeholders. Competences are obtained or developed during the process of learning by the student.

Learning outcomes are statements of what a learner is expected to know, understand and/or be able to demonstrate after completion of learning.

Competences represent a dynamic combination of knowledge, understanding, skills and abilities. Fostering competences is the object of educational programmes. Competences will be formed in various course units and assessed at different stages [9].

Tests, which are used to estimate students training level in a current academic discipline, are criterion-referenced (or should be such). Their goal is to reveal an examinee’s level of required knowledge, abilities and skills. Thus, the main goal of testing is establishing minimum number of points, achievement of which is enough to give a student a good mark.

A level of dynamic constituent parts of competence measurement is a difficult to create an algorithm and ambiguous in checking works problem. It needs to describe conceptual models, which are different to present.

Nowadays three dimensions model is certified and widespread.

First part of this model is a content, which is provides a content validity of a tool set, or to be more precise its compliance with educational programmes. Second part of the model are the process requirements – a type of test questions in a general case. Third part of the model is a level of a cognitive activity [5].

## COMMON PRINCIPLES OF TEST’S CONTENT SELECTION

Recommendations in educational tests in programming development are given in this work. Collected common principles of tests in programming creation are based and correlated with common principles of content selection.

Content selection common principles provide high content validity support. Content validity is the estimate of how much a measure represents every single element of a construct [6, 12, 5].

1. Representativeness principle. This establishes a procedure of test content selection to provide optimum completeness and correctness of test content proportions.
2. Significance principle. This orders to put on the test the most significant content items, which are connected with the key topic of the course. The key topic extraction needs course content to be structured before its putting on the test.



- System principle. This means that content items are put in order, and are connected with each other with a special hierarchy and a common knowledge structure. Following this principle test may be used not only to check educational achievements but to estimate knowledge structure of students' quality as well.

#### TYPES OF TEST QUESTIONS

Before describing tests in programming structure and separating questions types, which allowed estimating learning outcomes, we will give a brief test questions review. To make further narration more convenient we'll use symbols.

Test questions are divided into several types. Using different kinds of questions in a test can improve its quality and make it more flexible.

#### Question types:

- Multiple choice questions (MCQs) – student should choose one from a list possible answers.
- Multiple response questions (MRQs) – student should choose one or more from a list possible answers, one or more (even all) options can be the keys.
- Text/Numerical question (Short answer questions – SAQ) – student should input text, numbers or both into a special empty text field.
- Matching questions involve linking items in one list to items in a second list.
- True/False questions require a student to assess whether a statement is true or not.
- Author's type of questions [5, 6].
- etc. [2, 5, 6]

Traditionally a MCQ or a MRQ consists of:

- a stem – question text;
- options – the choices are given after the stem.

The correct answer (or answers for MRQs) in the options list is called “the key”. The incorrect (but verisimilar) answers in the options list are called “distracters” [2].

#### Example 1. MRQ question

**System class String static methods which are returned a string are:**

- Join()
- Equals()
- Copy()
- CompareOrdinal()
- Intern()

#### Example 2. MCQ question

**Compilation and running of this code:**

```
int dif;
char ch1 = 'A';
char ch2 = 'c';
dif=Char.ToUpper(ch2).CompareTo(ch1);
Console.Write(dif);
```

**will output:**

options

- 1) -1
- 2) 1
- 3) 2
- 4) 34
- 5) 0

distracters

key

#### Example 3. SAQ question

**Compilation and running of this code:**

```
int a = 7;
int i = 0;
while (a == 7)
{
    if (i < 4)
        Console.Write(i++);
    else
        Console.Write(++i);
    break;
}
```

**will output:** \_\_\_

#### TEST IN PROGRAMMING COMMON STRUCTURE

Test consists of a number of questions (test problems). Besides dividing test questions into listed classical question types it is desirable to use another one classification. Each test problem consists of question's body which can contain whole programs or code fragments.

#### Types and features of the test problems

A-type: Questions for checking a programming language syntax and semantic theoretical knowledge. This type of questions is represented by both the MCQs and MRQs. Questions of this type do not contain code of whole programs of all-in-one blocks of code.

B-type: Applied questions for checking practical skills; functionality skills analysis, and development programs according to a given functionality skills. This type of questions is represented by MCQs, MRQs or (the better one) SAQs. Questions of this type may contain program's code or all-in-one blocks of code.

#### B-type question's features

- Program functionality or all-in-one code block functionality analysis. Test problem is designed as MCQ. Student is asked to resolve what the code that is given in a stem execution result is. Source data are defined by the program or by a user. In the last case their values should be given in a stem. If a syntax error is intentionally added to the code the option is given (Example 2, 3).
- Analysis if a program meets requirement functionality. Test problem is designed as MCQ or MRQ. Student is provided with the whole description of a program or a code block purpose, which is given with gaps. One or more options

should provide requirement functionality if they are put in the gaps.

- Causes of departure from predefined behavior analysis. Test problem is designed as MCQ or MRQ. Stem contains a program (block of code) purpose description, a whole program or all-in-one code block and a result of its execution. Student finds out presence of deviation of the program's result from the predefinition. Student detects how far result of a program deviates from the predefinition one of defines changings which make program meeting requirement functionality.

Test questions which are connected with syntax errors in a program's code should be designed as MCQs or MRQs with concrete clear options. This means that the given options shouldn't contain compiler's messages. Questions like "Which message will be generated by a compiler as a result of compilation?" are also unacceptable.

Quantity of A- and B-type questions in a structure of a test should be balanced and can't be changed by a test-developer of in a concrete test. Topic's contents distribution by question's types is free and is defined by a test-developer.

#### Test decoration requirements

Test, for example, can be prepared as separate MS Word file and contain 30 - 40 items.

Questions should be formed as a table, left column for a number of question and keys. Key for a short-answer question is indicated as a value, for MCQs and MRQs numbers of right options are enumerated (see Table 1).

Each stem of MCQ or MRQ is followed by five options. For MRQs even all of the options can be keys.

A stem and the options shouldn't be more than 24 text-strings long.

TABLE 1. TEST QUESTIONS DECORATION (FRAGMENT)

9	<b>In the given code block which determines if a string length and digits sum in it are equal</b>
35	<pre>string str = Console.ReadLine(); int sum = 0; for(int i = 0; i &lt; str.Length - 1; i++)     if(str[i] &gt; '0' - 1 &amp;&amp; str[i] &lt; '9' + 1)         sum += str[i] - 0; Console.WriteLine(sum == str.Length);</pre> <p><b>programmer made mistakes:</b></p> <ol style="list-style-type: none"> <li>1) only 1.8 digits summs up</li> <li>2) a string char is addressed by index</li> <li>3) digits codes are summed up</li> <li>4) variable sum is called in for</li> <li>5) the last char of string isn't analyzed</li> </ol>
10	<b>Mark commands after adding which into the following code block execution will output 6</b>
1345	<pre>using System; class Program {     static void Main()     {         // TODO add your code here     } }</pre> <ol style="list-style-type: none"> <li>1) Console.WriteLine(12 &gt;&gt; 1);</li> <li>2) Console.WriteLine(2 &amp; 4);</li> <li>3) Console.WriteLine(5 ^ 3);</li> <li>4) Console.WriteLine(6   4);</li> <li>5) Console.WriteLine(7 &gt;&gt; 1 &lt;&lt; 1);</li> </ol>

11	<b>Compilation and running of this code</b>
1:0:True	<pre>string s1 = "Cat", s2 = "cat", s3 = "Cat"; Console.WriteLine(s1.CompareTo(s2) + ""); Console.WriteLine(s1.CompareTo(s3)); Console.WriteLine(":" + (s1 == s3));</pre> <p><b>will output:_____</b></p>

Tests which are developed according to given specification are suitable to be used in two cases:

- Paper-and-pencil testing (blanks);
- Computer testing.

The time to complete a test (ex. 30 - 40 questions) is limited (ex. from 40 to 60 minutes). Results are assigned using a binary scale (dichotomous appraisal plan).

SAQs of B-type can be redesigned into MCQs or MRQs for the computer testing needs.

#### THE RESEARCH

A group of 88 first-year students of Software engineering department was given an exam in a test-form after a first semester of the "Programming" academic subject. Test's questions were designed according to the structure was given above. Test consisted of 30 questions; time was limited by 30 minutes. This test we will call in short "the test" here and below. The test was given in a computer-based form.

Examinee's results were put in a table – response matrix. We used a dichotomous scale, for the items. Response matrix was used as a base in calculating primary scores and in visualizing the distribution of scores in a diagram (Fig. 1).

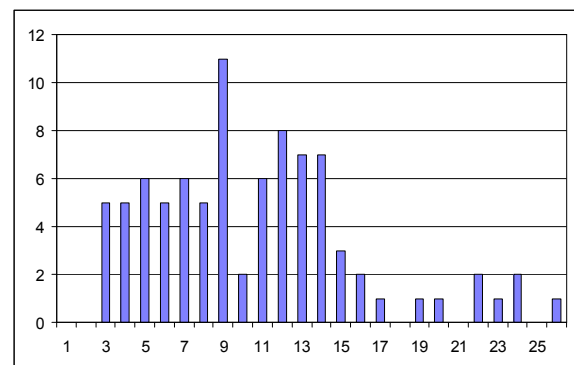


Figure 1

The mean is the average of all of the scores on the test. For the current test it was calculated as:

$$\bar{x} = \frac{\sum_{i=1}^N x_i}{N}$$

where  $x_i$  are the individual scores,  $N$  – a number of examinees.

There is a downfall in the middle of a diagram, near the mean ( $\bar{x} = 10, 47$ ). This means that we have got a bimodal deviation curve. Besides that we can draw a conclusion that all the examinees are divided into two level-groups. It seems to be quite reasonable because of peculiarity of the academic subject "Programming". It is learnt by the first-year undergraduates, former schoolchildren, whose base knowledge in programming is quite different. Some of them are lyceums-graduates and

well-trained in IT and programming, the others finished secondary schools and haven't got much special knowledge [3].

*Item analysis statistics*

The item difficulty (or item difficulty index) ( $p$ -value) is a measure of the proportion of examinees who answered the item correctly. The difficulty of an item  $j$  is calculated as:

$$p_j = \frac{N_j}{N}$$

where  $N_j$  – number of examinees with a score 1 on item  $j$ , and  $N$  – number of examinees [10].

TABLE 2. ITEM DIFFICULTY (FRAGMENT)

Item number	11	12	13	14	15	16	17
$p_j$	0,736	0,31	0,057	0,483	0,598	0,552	0,391

The middle  $p$ -value for our test is 0,353, minimum value is  $p_{13} = 0,057$ , maximum  $p_{11} = 0,736$ . The test hasn't got very difficult and very easy items. On the whole, the test is corresponded to the examinees' level.

The item discrimination index is a measure of how well an item is able to distinguish between examinees who are knowledgeable and who are not [10]. High values of this index correspond to good items and low – to bad ones. The item discrimination index for item  $j$  is calculated as:

$$D_j = \frac{n_j^b}{N^b} - \frac{n_j^w}{N^w}$$

where  $n_j^b$  – number of examinees from the best group with a score 1 on item  $j$ ,  $n_j^w$  – number of examinees from the worst group with a score 1 on item  $j$ ,  $N^b$  – number of examinees in the best group,  $N^w$  – number of examinees in the worst group.

In short, item discrimination indexes which are calculated for the test show that it works quite well for this scope of students. All of item discrimination indexes are positive, so better examinees did test better than worse examinees. Items 9, 10, 13, 28, 30 have  $D_j < 0,19$ . Items 1 and 22 have  $0,19 < D_j < 0,3$ . These items should be reviewed or altered.

The point-biserial correlation is the correlation between right (wrong) scores that students receive on a given item and the total scores that the examinees receive when summing up their scores across the remaining items [11]. The point-biserial correlation index is calculates as:

$$r_{pbjs} = \frac{(\mu_+^j - \mu_x)}{\sigma_x} \sqrt{p_j/q_j}$$

where  $\mu_+^j$  – the average total score for those students who answered item  $j$  correctly,  $\mu_x$  – the average total score for the whole group of examinees,  $\sigma_x$  – the standard deviation of the total score for the whole group of examinees,  $p_j$  – the difficulty index for item  $j$ ,  $q_j = 1 - p_j$  [12].

The point-biserial correlation indexes for the all test's items are positive (Table 2). High negative value means that examinees who scores well on the test have a lower probability of answering this item correctly.

TABLE 3. THE POINT-BISERIAL CORRELATION

Item number	1	2	3	4	5	6
$r_{pbis}$	0,405	0,286	0,223	0,34	0,269	0,585
Item number	7	8	9	10	11	12
$r_{pbis}$	0,448	0,333	0,038	0,225	0,263	0,473
Item number	13	14	15	16	17	18
$r_{pbis}$	0,383	0,549	0,409	0,385	0,575	0,366
Item number	19	20	21	22	23	24
$r_{pbis}$	0,596	0,572	0,619	0,324	0,456	0,526
Item number	25	26	27	28	29	30
$r_{pbis}$	0,572	0,562	0,374	0,29	0,539	0,282

The retest for the same examinees on this test is impossible. Thus the split-half method or the KR-20 may be used in estimating reliability of the test.

The KR-20 (Kuder-Richardson 20) was developed to handle only dichotomously scored items [12]. So far as we used dichotomous scale for the items of the test we apply the KR-20:

$$r_{KR-20} = \frac{N}{N-1} \left( 1 - \frac{\sum_{j=1}^N p_j q_j}{D_x} \right)$$

where  $N$  – number of test items,  $p_j$  – the difficulty index for item  $j$ ,  $q_j = 1 - p_j$ ,  $D_x$  – the total test variance.

For the test  $r_{KR-20} = 0,826$ .

Using the standard deviation of the total score ( $\sigma_x = 5,38$ ) and the reliability of the test we can calculate  $\sigma_E$  – standard error of measurement (SEM) to estimate how close to the true score the obtained score is [12].

$$\sigma_E = \sigma_x \sqrt{1 - r_{KR-20}} = 2,242$$

Prediction interval with the 5% level is:

$$(x_i - 2\sigma_E, x_i + 2\sigma_E) = (x_i - 4,484, x_i + 4,484)$$

According to the test theory and practice such reliability is quite acceptable and the test can be considered professional enough.

THE RESULTS SCALING

It will be recalled that the main goal of testing is establishing minimum number of points, achievement of which is enough to give a student a good mark.

The lifetime of study programs of the modern academic subjects is quite short. Besides that, groups of examined students are small. Usually they count one or two students' groups (25-30 persons in each), four-five groups are infrequent occurrence. Thus, there is impossible to get regulations which widely demonstrate a test quality and global scores of examinees, because of absence a huge representative sampling of examinees in the University.

It is also almost impossible to organize a peer review of all the tests, because a number of lecturers who are familiar with a content of each current test is limited and this kind of activity isn't traditionally contained into their syllabuses. The same troubles take place in setting a minimum number of points using a posteriori examination of a test and the results of it.

So, in spite of requirement of criterion-referenced type of interpretation of the test, we have to resort to norm-referenced type of interpretation. This gives a chance to determine an examinee's relative position within the specified group, to rank examinees according to their scores and to estimate indirectly difficulty of a test.

Reverting to the test, we used two approaches to interpret individual scores of the examinees.

First of all, we reduced individual scores to a standard Z-score. A mark of an examinee is calculated as:

$$Z_i = \frac{(x_i - \mu_x)}{\sigma_x}$$

where  $x_i$  – an individual score,  $\mu_x$  – the average total score for the whole group of examinees,  $\sigma_x$  – the standard deviation of the total score for the whole group of examinees.

z-values lies between -3 and +3, so they can't be directly use as marks in the ten-point score [3]. To transformation from z-values to ten-point values we use the following ratio:

$$B_i = \frac{10x_i(Z_{max} - Z_{min})}{(Z_i - Z_{min})}$$

where  $x_i$  – an individual score,  $Z_i$  – a mark in z-score,  $Z_{max}$  – maximum mark in z-score,  $Z_{min}$  – minimum mark in z-score.

For the test  $Z_{min} = -1,95$ ,  $Z_{max} = 2,89$ .

The threshold value for the ten-point score is 3,5. Marks less than this value are unsatisfactory, marks from 3,5 to 4,5 are satisfactory, from 4,5 to 7,5 are good, more than 7,5 – excellent [3].

To compare ten-point score with the traditional scores, we made a transformation from z-values to t-score:

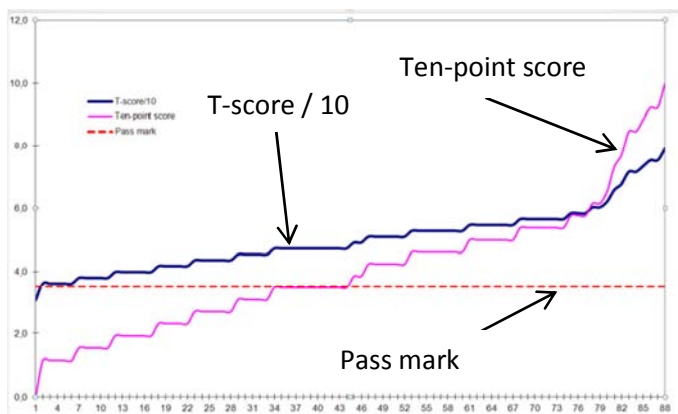


Figure 2. Total scores

$$T_i = 50 + 10Z_i$$

Limit values of t-score marks are 30,54 and 78,89. To make comparison easier results of calculating is put into a graph which is given on Figure 2. T-score values on Figure 2 were decreased in ten times.

#### CONCLUSION

Computer testing in “Programming” and “Algorithmic languages” is an effective method of impartial assessment

of student's achievement at the different stages of educational process. Testing is reasonable in preliminary examination to find out groups of students whose further education needs additional consultations or adaptation courses. Testing can provide self-control in learning academic disciplines which are connected with programming. Testing is effective as intermediate test-check and very useful then using as a part of total test-check, exam for example.

It should be noted that some factors prevent from regular and multi-faced usage of computer testing in institutions of higher education.

- Test development is a high work content activity, which isn't taken in consideration in lecturers' syllabuses.
- Occurring everywhere absence of licensing computer testing software with a special functionality to collect individual scores of examinees and items.
- Neediness of tests' approbation. Primary contingent of examinees is absent (tests are prepared to small groups of students; they can't be approved before their application). There are no administrative facilities to attract experts to analyze tests' questions and to interpret results.

Recommendations in development of educational tests in programming which are given in this work can be used as based ones in different universities where disciplines which are connected with algorithmic languages and programming are taught.

#### REFERENCES

- [1] H. Gulliksen, “Theory of mental tests”, New York: John Willey & Sons, Inc., 1950.
- [2] C. McKenna, J. Bull, “Designing effective objective test questions: an introductory workshop”, CAA Centre, June 1999
- [3] V.V. Podbelskiy, O.V. Maksimenkova. “Programming as a part of the Software Engineering education” // Proceedings of the 4-th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2010), 2010. 165 – 168 pp.
- [4] T. Dawson, “Basic concepts in classical test theory: relating variance partitioning in substantive analyses to the same process in measurement analyses”, URL: <http://www.eric.ed.gov/PDFS/ED406443.pdf>
- [5] В.И. Звонников, М.Б. Чельшкова. Современные средства оценивания результатов обучения. – М.: Издательский центр «Академия», 2009. – 224 с.
- [6] В.И. Звонников, М.Б. Чельшкова. Контроль качества обучения при аттестации: компетентностный подход. – М.: Университетская книга; Логос, 2010. – 272 с.
- [7] Sun Haiyang “An application of Classical test theory and Manyfacet Rasch measurement in analyzing the reability of an English test for non-English major graduates”, Chinese Journal of applied linguistics (Bimonthly). China, vol. 33, pp. 87 – 102, April 2010.
- [8] Т.П. Хлопова, Т.Л. Шапошникова, М.Л. Романова, А.Р. Ушаков. Математические модели дидактического процесса // Научно-теоретический журнал «Ученые записки». – 2010 – № 2. с. 107 – 112.
- [9] J. Gonzalez, R. Wagenaar, “Universities' contribution to the Bologna Process. An introduction”, Spain: Publicaciones de la Universidad de Deusto, 2008.
- [10] URL: [http://www.proftesting.com/test\\_topics/pdfs/steps\\_9.pdf](http://www.proftesting.com/test_topics/pdfs/steps_9.pdf)

- [11] S. Varma “Preliminary item statistics using point-biserial correlation and P-values”, URL: [http://www.eddata.com/resources/publications/EDS\\_Point\\_Biserial.pdf](http://www.eddata.com/resources/publications/EDS_Point_Biserial.pdf)
- [12] P.V. Engelhardt “An Introduction to Classical Test Theory as Applied to Conceptual Multiple-choice Tests”

# To The Parallel Composition of Timed Finite State Machines

Olga Kondratyeva, Maxim Gromov

Radiophysics Faculty  
Tomsk State University  
Tomsk, Russia

kondratyeva.olga.vic@gmail.com, gromov@sibmail.com

**Abstract**—This paper deals with the problem of the parallel composition construction for two Timed Finite State Machines (TFSMs). As a key to the solution of this problem we use parallel composition of common Finite State Machines (FSMs). We transform given TFSMs to FSMs and prove theorem, that obtained FSMs correctly describe behaviour of the given TFSMs. Then we build parallel composition of these FSMs, which being transformed back to TFSM, gives desired parallel composition of the given TFSMs

**Keywords**—Finite State Machine; Timed Finite State Machine; parallel composition

## I. INTRODUCTION

The Timed Finite State Machine (TFSM) is a model based on well-known Finite State Machine (FSM), which allows explicit description of a time aspects of system behaviour. For example, reaction of a system can be different depending on the time moment an input action is applied to it. In the last few years the interest to the various problems of TFSM has increased. The main lines of researches covered by the post papers are the analysis problems: relations between TFSMs [1, 2] and test generation methods against those relations [3, 4].

In our paper we consider a problem of synthesis, namely the problem of parallel composition construction of two TFSMs. This procedure gives an instrument to build complex systems from simple ones, each described by a TFSM. Also, the approach we used in this paper to describe a parallel composition construction procedure opens the way for solving various problems of TFSMs.

## II. PRELIMINARIES

In this section we give some notions and definitions, which we shall use all over the paper.

### A. Language

An *alphabet* is a finite non-empty set of symbols and as usual, given an alphabet  $X$ , we denote  $X^*$  the set of all finite sequences (words) of symbols from  $X$  including the empty word  $\varepsilon$ . The number of symbols in a sequence we shall call *length* of this sequence; by definition, length of the empty word is zero. A subset  $L \subseteq X^*$  is a *language* over alphabet  $X$ .

Let language  $L$  be defined over alphabet  $Y$  and  $X$  be a non-empty subset of  $Y$ . The  $X$ -restriction  $L \downarrow_X$  of the language  $L$  is derived by deleting from each sequence of  $L$  each symbol of the set  $\setminus X$ . When the language  $L$  is defined over alphabet  $X$ , and  $Y$  is some alphabet that is disjoint with  $X$ , consider the mapping  $\varphi: X \rightarrow 2^{(X \cup Y)^*}$  such, that  $\varphi(x) = \{\alpha x \beta : \alpha, \beta \in Y^*\}$ . This mapping can be extended over sequences from  $X^*$  as follows. Let  $\gamma$  be a sequence from  $X^*$  and  $x$  be a symbol from  $X$ , then  $\varphi(\varepsilon) = Y^*$  and  $\varphi(x\gamma) = \varphi(x) \cdot \varphi(\gamma)$ , where the sign “ $\cdot$ ” stands for concatenation of sequences. We shall call the language  $L \uparrow_Y = \{\varphi(\gamma) : \gamma \in L\}$  the  $Y$ -expansion of language  $L$ .

### B. Finite automata

There exists a special set of languages which can be described by the use of finite automata; those are regular languages, which are closed under union, concatenation, complementation, intersection and also under restriction and expansion.

A *finite automaton* (FA) is a 5-tuple  $\mathbf{S} = \langle S, A, s_0, \delta_S, Q \rangle$ , where  $S$  is a non-empty finite set of states with the designated initial state  $s_0$ ,  $A$  is a finite alphabet of actions,  $\delta_S \subseteq S \times A \times S$  is a transition relation, and  $Q \subseteq S$  is a set of final (accepting) states. If  $(s_1, a, s_2) \in \delta_S$ , then we say, that automaton  $\mathbf{S}$  in the state  $s_1$  takes action  $a$ , and changes its state to the state  $s_2$ ; the state  $s_2$  is called an  $a$ -successor of the state  $s_1$  and we denote by  $suc_S(s_1, a)$  the set of all  $a$ -successors of the state  $s_1$ . Function  $suc_S$  can be extended over sequences from  $A^*$  as follows:

$$suc_S(s_1, \beta a) = \{suc_S(s_2, a) : s_2 \in suc_S(s_1, \alpha)\}.$$

By the definition  $suc_S(s_1, \varepsilon) = s_1$ .

Finite automaton  $\mathbf{S}$  is called *deterministic* if for each pair  $(s_1, a) \in S \times A$  there is at most one state  $s_2 \in S$  such that  $(s_1, a, s_2) \in \delta_S$ , i.e.  $|suc_S(s_1, a)| \leq 1$ , otherwise, the finite automaton is *non-deterministic*.

Finite automaton  $\mathbf{S}$  is called *complete* if for each pair  $(s_1, a) \in S \times A$  there is at least one state  $s_2 \in S$  such that  $(s_1, a, s_2) \in \delta_S$ , i.e.  $|suc_S(s_1, a)| \geq 1$ , otherwise, the finite automaton is *partial*.

Let us consider a word  $\beta \in A^*$ . Automaton  $\mathbf{S}$  *recognizes* or *accepts*  $\beta$  if there exists an accepting state  $q \in Q$  such that  $q$  is a  $\beta$ -successor of the initial state, i.e.  $q \in suc_S(s_0, \beta)$ . The

---

This paper is partially supported by Russian Foundation for Basic Research (Grant 10-08-92003-HHC\_a)

set  $L_S$  of all sequences, which are accepted by  $\mathbf{S}$ , is the *language accepted* by the automaton or simply the *language of the automaton*  $\mathbf{S}$ . The language of a finite automaton is a regular language [5].

### C. Finite State Machines

To describe behaviour of a system, which transforms sequences over one (input) alphabet into sequences over another (output) alphabet, special kind of automata, called Finite State Machine, is usually used [6].

A *finite state machine* (FSM) is a 5-tuple  $\mathbf{S} = \langle S, I, O, s_0, \lambda_S \rangle$ , where  $S$  is a non-empty finite set of states with initial state  $s_0$ ,  $I$  and  $O$  are disjoint finite input and output alphabets,  $\lambda_S \subseteq S \times I \times O \times S$  is the transition relation. If  $(s_1, i, o, s_2) \in \lambda_S$ , then we say, that the FSM  $\mathbf{S}$  in the state  $s_1$  gets the input action  $i$ , produces the output action  $o$  and changes its state to  $s_2$ ; the state  $s_2$  is called an *i/o-successor* of the state  $s_1$ . The set of all *i/o*-successors of the state  $s_1$  is denoted  $suc_S(s_1, i, o)$ , while

$$suc_S(s_1, i) = \{s_2 \in S : \exists o \in O \text{ such that } s_2 \in suc_S(s_1, i, o)\}$$

is the set of all *i*-successors of the state  $s_1$ .

Functions  $suc_S(s_1, i, o)$  and  $suc_S(s_1, i)$  can be extended to the sequences  $\alpha \in I^*$  and  $\beta \in O^*$ , where lengths of  $\alpha$  and  $\beta$  are equal, as follows:

$$suc_S(s_1, \alpha i, \beta o) = \{suc_S(s_2, i, o) : s_2 \in suc_S(s_1, \alpha, \beta)\}$$

and

$$suc_S(s_1, \alpha i) = \{suc_S(s_2, i) : q \in suc_S(s_1, \alpha)\}.$$

By the definition  $suc_S(s_1, \varepsilon) = suc_S(s_1, \varepsilon, \varepsilon) = s_1$ .

FSM  $\mathbf{S}$  is *deterministic* if for each pair  $(s_1, i) \in S \times I$  there is at most one pair  $(o, s_2) \in S \times O$  such that  $(s_1, i, o, s_2) \in \lambda_S$ , i.e.  $|suc_S(s_1, i)| \leq 1$ , otherwise, FSM  $\mathbf{S}$  is *non-deterministic*.

FSM  $\mathbf{S}$  is *complete* if for each pair  $(s_1, i) \in S \times I$  there is at least one pair  $(o, s_2) \in S \times O$  such that  $(s_1, i, o, s_2) \in \lambda_S$ , i.e.  $|suc_S(s_1, i)| \geq 1$ , otherwise, FSM  $\mathbf{S}$  is *partial*.

FSM  $\mathbf{S}$  is *observable* if for each triple  $(s_1, i, o) \in S \times I \times O$  there is at most one state  $s_2 \in S$  such that  $(s_1, i, o, s_2) \in \lambda_S$ , i.e.  $|suc_S(s_1, i, o)| \leq 1$ , otherwise, FSM  $\mathbf{S}$  is *non-observable*.

A sequence  $\sigma = (i_1, o_1)(i_2, o_2) \dots (i_n, o_n) \in (I \times O)^*$  is called a *trace* of given FSM  $\mathbf{S}$  if the set of  $\alpha/\beta$ -successors, where  $\alpha = i_1 i_2 \dots i_n$  and  $\beta = o_1 o_2 \dots o_n$ , of the initial state of  $\mathbf{S}$  is non-empty, i.e.  $suc_S(s_0, \alpha, \beta) \neq \emptyset$ . The set of all traces of the FSM is the *language*  $L_S$  of the FSM  $\mathbf{S}$ . Further, talking about traces of an FSM, we assume that a sequence  $\sigma$  and the corresponding pair  $\alpha/\beta$  are equivalent notions.

Given FSM  $\mathbf{S} = \langle S, I, O, s_0, \lambda_S \rangle$ , the automaton  $Aut(\mathbf{S})$  is a 5-tuple  $\langle S \cup (S \times I), I \cup O, s_0, \delta_S, S \rangle$ , where for each transition  $(s_1, i, o, s_2) \in \lambda_S$  there are two transitions  $(s_1, i, (s_1, i))$ , and  $((s_1, i), o, s_2) \in \delta_S$ . Since the language  $L_S^{aut}$  of the automaton  $Aut(\mathbf{S})$  is the language of the FSM [7] it

holds that  $L_S^{aut} \subseteq (IO)^*$ , where  $IO$  is concatenation of alphabets  $I$  and  $O$ .

### D. Timed Finite State Machines

A *timed finite state machine* (TFSM) is a 6-tuple  $\mathbf{S} = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$ , where the 5-tuple  $\langle S, I, O, s_0, \lambda_S \rangle$  is an FSM and  $\Delta_S: S \rightarrow S \times (\mathbb{N} \cup \{\infty\})$  is a time-out function. If  $\Delta_S(s_1) = (s_2, n)$ , then the TFSM  $\mathbf{S}$  in the state  $s_1$  will wait an input action for  $n$  time units (ticks), and if none arrives it will move to the state  $s_2$  (possibly the same as  $s_1$ ), without producing any output. If  $\Delta_S(s_1) = (s_2, \infty)$ , then we require  $s_2 = s_1$  and the TFSM can stay in the state  $s_1$  infinitely long, waiting for an input. Definitions of a deterministic, complete and observable TFSM are based on the corresponding definitions for underlying FSM.

A special timed or clock variable can be associated with a TFSM; this variable counts time ticks passed from the moment when the last transition has been executed and is reset to 0 after each transition (input-output or time-out). In this paper, for the sake of simplicity, we assume that the output is produced immediately after a machine gets an input, i.e., we do not consider delays when executing transitions.

A pair  $(i, t) \in I \times (\mathbb{N} \cup \{0\})$  is a *timed input* meaning that the input  $i$  is submitted to the TFSM  $t$  ticks later than the previous output has been produced. A sequence of inputs is a timed input sequence.

We also define a special function  $time_S$  [1] as follows:

1.  $time_S(s, t) = s$  for all  $t \in \mathbb{N} \cup \{0\}$  if  $\Delta_S(s) = (s, \infty)$ .
2.  $time_S(s_1, t) = s_1$  for all  $t < T$  and  $\Delta_S(s_1) = (s_2, T)$ .
3.  $time_S(s_1, t) = s_2$  for  $t = T$  and  $\Delta_S(s_1) = (s_2, T)$ .
4. for  $t > T$  and  $\Delta_S(s_1) = (s_2, T)$  define recursively  $time_S(s_1, t) = time_S(s_2, t - T)$ , i.e. there is a sequence  $s_1, s_2, \dots, s_k$  such that for each  $j = 1 \dots k - 1$  it holds  $\Delta_S(s_j) = (s_{j+1}, T_j)$  and  $T_1 + T_2 + \dots + T_{k-1} \leq t < T_1 + T_2 + \dots + T_{k-1} + T_k$ , then  $time_S(s_1, t) = s_k$ .

The function  $suc_S$  is defined similar to that defined for an FSM and is extended to timed inputs as follows:  $suc_S(s, (i, t), o) = suc_S(time_S(s, t), i, o)$ .

A sequence

$$\sigma = (i_1, t_1, o_1)(i_2, t_2, o_2) \dots (i_n, t_n, o_n) \in [I \times (\mathbb{N} \cup \{0\}) \times O]^*$$

is called a *functional trace* of a TFSM  $\mathbf{S}$ , if the following holds  $suc_S(s_0, \alpha, \beta) \neq \emptyset$ , where  $\alpha = (i_1, t_1)(i_2, t_2) \dots (i_n, t_n)$  and  $\beta = o_1 o_2 \dots o_n$ . The set  $L_S$  of all functional traces of the TFSM  $\mathbf{S}$  is the *f-language of the TFSM*  $\mathbf{S}$ . Here we again assume, that the pair  $\alpha/\beta$  and the sequence  $\sigma$  are the equivalent notions, when speaking about f-language of a TFSM.

### E. Equivalence of automata, FSMs and TFMSs

Two finite automata  $\mathbf{S}$  and  $\mathbf{P}$  with languages  $L_S$  and  $L_P$  are said to be *equivalent* if  $L_S = L_P$ .



Two FSMs  $\mathbf{S}$  and  $\mathbf{P}$  with languages  $L_S$  and  $L_P$  are said to be *equivalent* if  $L_S = L_P$ .

Two TFSMs  $\mathbf{S}$  and  $\mathbf{P}$  with f-languages  $L_S$  and  $L_P$  are said to be *equivalent* if  $L_S = L_P$ .

### III. PARALLEL COMPOSITION

In this paper we propose definition of parallel composition for two TFSMs. This definition relies on the definition of FSM parallel composition and the latter is defined in terms of parallel composition of corresponding automata. For that reason we also describe the conversion procedure [8] of a TFSM into an FSM, which then is used for the parallel composition construction. We also prove, that built FSM correctly reflects the language of a given TFSM.

#### A. Parallel composition of languages

Given pairwise disjoint alphabets  $X, Y, Z$ , languages  $L_1$  over  $X \cup Y$  and  $L_2$  over  $Y \cup Z$ , the *parallel composition* of languages  $L_1$  and  $L_2$  is the language

$$L = [(L_1)_{\uparrow Z} \cap (L_2)_{\uparrow X}]_{\downarrow X \cup Z}$$

defined over  $X \cup Z$  and denoted  $L_1 \diamond_{X \cup Z} L_2$  or just  $L_1 \diamond L_2$  when the union  $X \cup Z$  is clear from context.

#### B. Parallel composition of automata

Given two finite automata  $\mathbf{S} = \langle S, I \cup U, s_0, \delta_S, Q_S \rangle$  and  $\mathbf{P} = \langle P, U \cup O, p_0, \delta_P, Q_P \rangle$ , the automaton  $\mathbf{C} = \langle C, I \cup O, c_0, \delta_C, Q_C \rangle$  is a *parallel composition of automata*  $\mathbf{S}$  and  $\mathbf{P}$ , denoted  $\mathbf{C} = \mathbf{S} \diamond \mathbf{P}$ , iff  $L_C = L_S \diamond L_P$ . To obtain composition of automata define expansion and restriction over automata as follows.

Given disjoint alphabets  $I$  and  $O$  and an automaton  $\mathbf{S} = \langle S, I, s_0, \delta_S, Q_S \rangle$ .  $O$ -*expansion* of  $\mathbf{S}$  is an automaton  $\mathbf{S}_{\uparrow O} = \langle S, I \cup O, s_0, \delta_S \cup \mu_S, Q_S \rangle$ , where  $\mu_S \subseteq S \times O \times S$  contains all triples  $(s, o, s)$ , such that  $o \in O$  and  $s \in S$ , i.e. to expand an automaton we add loops marked with all symbols of alphabet  $O$  for each state.

Given disjoint alphabets  $I$  and  $O$  and an automaton  $\mathbf{S} = \langle S, I \cup O, s_0, \delta_S, Q_S \rangle$ .  $I$ -*restriction* of  $\mathbf{S}$  is an automaton  $\mathbf{S}_{\downarrow I} = \langle S, I \cup \{\tau\}, s_0, \mu_S, Q_S \rangle$ , where for each transition  $(s_1, a, s_2) \in \delta_S$  we add transition  $(s_1, a, s_2)$  into  $\mu_S$  in case  $a \in I$ , while we add transition  $(s_1, \tau, s_2)$  into  $\mu_S$  in case  $a \in O$ , i.e. to restrict an automaton we replace all symbols of alphabet  $O$  by special symbol  $\tau$ . An automaton without  $\tau$ -moves can be derived by the determinization procedure [9].

Now, the procedure of parallel composition construction of two given automata  $\mathbf{S}$  and  $\mathbf{P}$  can be described by the formula:

$$\mathbf{C} = [\mathbf{S}_{\uparrow O} \cap \mathbf{P}_{\downarrow I}]_{\downarrow I \cup O}$$

#### C. Parallel composition of FSMs

Following [7], we define the parallel composition of two FSMs (Fig. 1) based on their corresponding automata. However, the language of the parallel composition of two automata is not necessary an FSM language. For this reason,

the obtained language should be intersected with the language  $(IO)^*$ , where  $I$  and  $O$  are external input and output alphabets of composition, to ensure that each input is followed by some output.

Given FSMs  $\mathbf{S} = \langle S, I_1 \cup V, U \cup O_1, s_0, \lambda_S \rangle$  and  $\mathbf{P} = \langle P, I_2 \cup U, V \cup O_2, p_0, \lambda_P \rangle$ , the parallel composition  $\mathbf{C} = \mathbf{S} \diamond \mathbf{P}$  is derived in the following way. We first derive corresponding automata  $Aut(\mathbf{S})$  and  $Aut(\mathbf{P})$  and the parallel composition  $Aut(\mathbf{S}) \diamond Aut(\mathbf{P})$ . The obtained automaton then is intersected with the automaton that accepts the language  $(IO)^*$  and is transformed to the FSM  $\mathbf{C}$  coupling inputs with the following outputs. FSM  $\mathbf{C} = \langle C, I, O, c_0, \lambda_C \rangle$  is the parallel composition of  $\mathbf{S}$  and  $\mathbf{P}$ , where  $I = I_1 \cup I_2$  and  $O = O_1 \cup O_2$ .

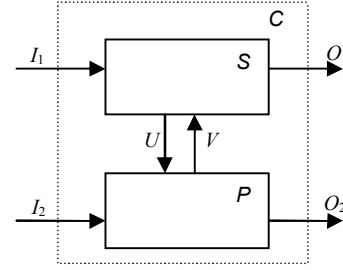


Figure 1 – Parallel composition of FSMs  $\mathbf{S}$  and  $\mathbf{P}$ .

It is proven [7], that parallel composition describes following interaction of composed FSMs  $\mathbf{S}$  and  $\mathbf{P}$  (Figure 1). The system starts its work, when both  $\mathbf{S}$  and  $\mathbf{P}$  are in their initial states, i.e. composition  $\mathbf{C} = \mathbf{S} \diamond \mathbf{P}$  is in its initial state. External environment applies input action either on channel  $I_1$  or  $I_2$ , but only one at a time, and then waits for an external output reaction of the system through the one of the output channels  $O_1$  or  $O_2$ . The component FSM, which just have got an input action, processes this input and produces either an external output (and so external environment can apply its next input action), or an internal output action that is internal input action for another component FSM. In the latter case, the second component FSM processes a submitted internal input and produces either an external output or an internal output applied to the first component FSM. The dialog between component FSMs continues until one of them produces an external output. When an external output is produced the system is ready to accept the next external input. Here we notice that there can be an external input initiating an infinite dialog between component FSMs. Such infinite cycles of internal actions are called livelocks. However, in practical situations, except of some special cases, input sequences inducing livelocks are usually forbidden.

#### D. Correspondence between Timed Finite State Machine and Finite State Machine

Before we propose how to construct the parallel composition of timed finite state machines, we introduce the transformation procedure of a TFSM into an FSM and back, and then prove, that obtained FSM correctly describes f-language of the TFSM.



Given TFMSM  $\mathbf{S} = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$ , we can build an FSM with similar set of functional traces by adding designated input  $1 \notin I$  and output  $N \notin O$  [8]. Corresponding FSM  $\mathbf{A}_S = \langle S \cup S_t, I \cup \{1\}, O \cup \{N\}, s_0, \lambda_S^\Delta \rangle$  can be built by adding  $T-1$  copies for each state  $s \in S$  with defined a finite time-out  $T > 1$ . There is a chain of transitions between these copies marked with special input-output symbol  $1/N$ . All other transitions are preserved for each copy. Formally, constructing of  $\mathbf{A}_S$  can be done by the use of the following rules:

1.  $S_t$  contains all such states  $\langle s, t \rangle$ ,  $t = 1, \dots, T-1$  where  $s \in S$  and  $\Delta_S(s) = (s', T)$ ,  $1 < T < \infty$ .
2. For each  $s \in S$  and  $\langle s, t \rangle \in S_t$  and for each  $i/o$ ,  $i \in I$ ,  $o \in O$ , there are transitions  $(s, i, s', o)$ ,  $(\langle s, t \rangle, i, s', o) \in \lambda_S^\Delta$  iff there is a transition  $(s, i, s', o) \in \lambda_S$ .
3. For each  $s \in S$  such that  $\Delta_S(s) = (s, \infty)$  there is a transition  $(s, 1, s, N) \in \lambda_S^\Delta$ .
4. For each  $s \in S$  such that  $\Delta_S(s) = (s', T)$ ,  $T = 1$ , there is a transition  $(s, 1, s', N) \in \lambda_S^\Delta$ .
5. For each  $s \in S$  such that  $\Delta_S(s) = (s', T)$ ,  $1 < T < \infty$ , there are transitions  $(s, 1, \langle s, 1 \rangle, N) \in \lambda_S^\Delta$ ; for each  $j = 1, \dots, T-2$  there are transitions  $(\langle s, j \rangle, 1, \langle s, j+1 \rangle, N) \in \lambda_S^\Delta$  and  $(\langle s, T-1 \rangle, 1, s', N) \in \lambda_S^\Delta$ .

We use  $S_A$  to denote  $S \cup S_t$ ,  $I_A$  to denote  $I \cup \{1\}$  and  $O_A$  to denote  $O \cup \{N\}$ .

By construction, when a given TFMSM  $\mathbf{S}$  has  $n$  states, a corresponding FSM  $\mathbf{A}_S$  has  $\sum_{s \in S} n(s)$  states, where  $n(s) = 1$  for  $\Delta_S(s) = (s, \infty)$  and  $n(s) = T$  for  $\Delta_S(s) = (s', T)$ .

Consider an example in Figure 2. State  $q$  of TFMSM has timeout 2 and therefore, we add one copy of  $\langle q, 1 \rangle$  (denoted " $q1$ ") which is  $1/N$ -successor of the state  $q$  while its  $1/N$ -successor is  $s$ . The sets of successors of  $q$  and  $\langle q, 1 \rangle$  for all other I/O pairs coincide.

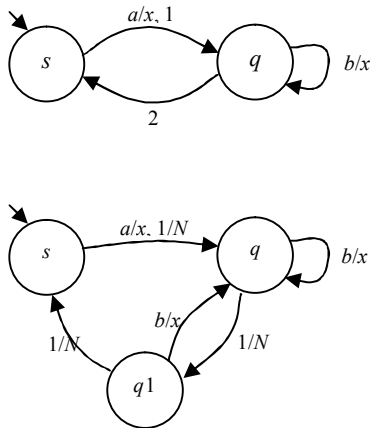


Figure 2 – TFMSM  $\mathbf{S}$  (top figure) and corresponding FSM  $\mathbf{A}_S$  (bottom figure)

An FSM  $\mathbf{A}_S = \langle S_A, I_A, O_A, s_0, \lambda_S^\Delta \rangle$  such that there are no transitions marked with  $1/o$  or  $i/N$  where  $o \neq N$  and  $i \neq 1$  can be transformed to TFMSM  $\mathbf{S} = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$  using the following rules:

1.  $\Delta_S(s) = (s, \infty)$  iff  $(s, 1, s, N) \in \lambda_R^\Delta$ .
2. Define  $\Delta_S(s) = (s_T, T)$  for all such  $s$  that there is a chain of transitions  $s \xrightarrow{1/N} s_1 \xrightarrow{1/N} \dots \xrightarrow{1/N} s_{T-1} \xrightarrow{1/N} s_T$ ,  $s, s_T \in S_A$ ,  $T \geq 1$  and for each  $i/o \in I \times O$  and  $1 \leq j \leq T-1$  it holds that  $\text{suc}_S^\Delta(s_j, i, o) = \text{suc}_S^\Delta(s, i, o)$ , but for some  $i/o \in I \times O$  it holds that  $\text{suc}_S^\Delta(s, i, o) \neq \text{suc}_S^\Delta(s_T, i, o)$ .
3. For each  $s \in S_A$ ,  $i \in I$  and  $o \in O$ , if  $(s, i, s', o) \in \lambda_S^\Delta$  then  $(s, i, s', o) \in \lambda_R$ .

Notice that transformation from a given TFMSM to an FSM according to the above rules is unique whereas the back transformation from an FSM to a TFMSM could be made in different ways; however all such TFMSMs are pairwise equivalent, i.e. their f-languages are the same (see the Corollary 2 to Proposition 1).

The following statements establish the relationship between a TFMSM and the corresponding FSM built by the above rules.

**Proposition 1.** FSM  $\mathbf{A}_S$  has a trace  $\underbrace{1/N \dots 1/N}_t i_1/o_1 \dots \underbrace{1/N \dots 1/N}_m i_m/o_m$  iff TFMSM  $\mathbf{S}$  has a functional trace  $\langle i_1, t_1 \rangle / o_1 \dots \langle i_m, t_m \rangle / o_m$ .

**Proof.** According to the rules of constructing  $\mathbf{A}_S$ , for each two states  $s_1$  and  $\langle s_1, j \rangle$  and each  $i \in I$  and  $o \in O$ , the set of  $i/o$ -successors of  $\langle s_1, j \rangle$  coincides with the set of  $i/o$ -successors of state  $s_1$  in  $\mathbf{S}$ . Thus if there exists such  $s_2$  that  $(s_1, i, s_2, o) \in \lambda_S$ , then there is a transition  $s_1 \xrightarrow{i/o} s_2$  in both machines  $\mathbf{S}$  and  $\mathbf{A}_S$  and there is a transition  $\langle s_1, j \rangle \xrightarrow{i/o} s_2$  in  $\mathbf{A}_S$ . Therefore, it is enough to show that  $\mathbf{A}_S$  is moving from state  $s_1$  to some state  $q \in S \cup S_t$  under the sequence  $\underbrace{1/N \dots 1/N}_t$  with  $t > 0$ , and the set of  $i/o$ -successors of  $q$  in  $\mathbf{A}_S$  coincides with the set of  $i/o$ -successors of the state  $\text{time}_S(s_1, t)$  in  $\mathbf{S}$ .

If  $\Delta_S(s_1) = (s_1, \infty)$ , then  $\text{time}_S(s_1, t) = s_1$  holds for each value of  $t$ ; therefore, there is a transition  $(s_1, 1, s_1, N) \in \lambda_S^\Delta$  and  $\mathbf{A}_S$  remains at state  $s_1$  under the sequence  $\underbrace{1/N \dots 1/N}_t$ .

Consider now  $\Delta_S(s_1) = (s_T, T)$ . If  $t < T$ , then  $\text{time}_S(s_1, t) = s_1$  and the sequence  $\underbrace{1/N \dots 1/N}_t$  moves  $\mathbf{A}_S$  from

$s_1$  to  $\langle s_1, t \rangle$ . The set of  $i/o$ -successors of  $\langle s_1, t \rangle$  in  $A_S$  coincides with the set of  $i/o$ -successors of  $s_1$  in  $S$ . If  $t \geq T$ , then  $time_S(s_1, t) = time_S(time_S(s_1, T), t - T) = time_S(s_T, t - T)$ . By construction, the sequence  $\frac{1/N \dots 1/N}{T}$  moves  $A_S$  from  $s_1$  to  $s_T$ , and the sequence  $\frac{1/N \dots 1/N}{t-T}$  is applied to  $A_S$  at state  $s_T$ , i.e., this case is inductively reduced to the previous case  $t < T$ .  $\square$

**Corollary 1.** TFSMs  $S$  and  $P$  are equivalent iff corresponding FSMs  $A_S$  and  $A_P$  are equivalent.

**Corollary 2.** If TFSMs  $S$  and  $P$  both are built by the above procedure from an FSMs  $A_S$ , then  $S$  and  $P$  are equivalent.

**Proposition 2.** TFSM  $S$  is deterministic (complete or observable) iff the corresponding FSM  $A_S$  is deterministic (complete or observable).

**Proof.** The property to be deterministic, observable and complete is specified by the cardinality of sets of  $i/o$ - and  $i$ -successors. FSM  $A_S$  has one and only one transition with pair  $1/N$  at each state, that is why properties of FSM  $A_S$  to be deterministic, observable and complete depend on transitions with other I/O pairs.

By construction it holds that  $suc_S^\Delta(\langle s, t \rangle, i, o) = suc_S^\Delta(s, i, o) = suc_S(s, i, o)$  for each state  $s$ ,  $\Delta_S(s) = (s', T)$ , and for any value of  $t < T$ . Hence  $|suc_S^\Delta(\langle s, t \rangle, i, o)| = |suc_S^\Delta(s, i, o)| = |suc_S(s, i, o)|$  and  $|suc_S^\Delta(\langle s, t \rangle, i)| = |suc_S^\Delta(s, i)| = |suc_S(s, i)|$ .  $\square$

*E. Parallel composition of TFSMs*

Parallel composition of two TFSMs  $S$  and  $P$  is a TFSM  $C = S \diamond P$  obtained from the FSM  $A_S \diamond A_P$ .

Let us illustrate our approach by constructing the parallel composition of TFSMs.

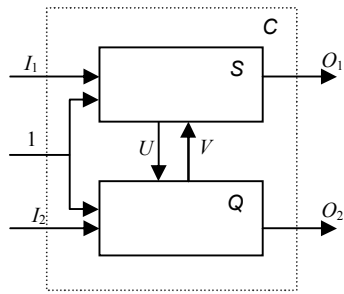


Figure 3 - Parallel composition of TFSMs as a parallel composition of corresponding FSMs

The parallel composition of FSMs that corresponds to the parallel composition of TFSMs is shown in Figure 3. In this case, port 1 is a common port for both machines as it corresponds to a counter of ticks and this accepts the designated input 1 that is an input for both component FSMs and can be considered as an input that synchronizes time

behaviour of component FSMs. The designated output  $N$  is observed, when there are no outputs at ports  $O_1$  and  $O_2$  (it is observed at both of the ports). Each component FSM has its own time variable, which increments every moment when component gets the designated input 1, and since this signal is applied via a common port for both components the global time is used, and thus, we can say that it synchronizes the behaviour of component FSMs.

As an example, consider the composition of TFSM  $S$  in Fig. 2 and  $P$  in Fig. 4 where corresponding FSMs are shown as bottom figures. Consider symbols  $a$  and  $o$  to be external input and output respectively,  $x$  and  $b$  are internal symbols.

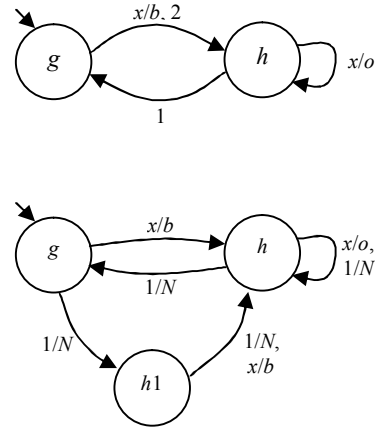


Figure 4 – TFSM  $P$  (top figure) and corresponding FSM  $A_P$  (bottom figure)

To derive the parallel composition of FSMs, we firstly construct the related automata which are shown in Figure 5. Double lines denote accepting states.

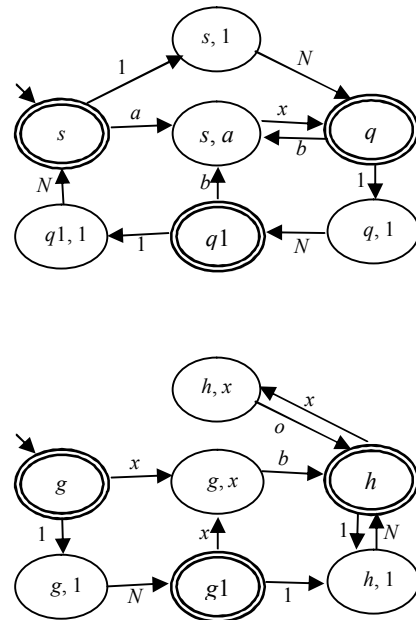


Figure 5 – Automata  $Aut(A_S)$  (top figure) and  $Aut(A_P)$  (bottom figure)

The second step is to derive the intersection of expended automata that is shown in Figure 6. This intersection should be restricted onto external alphabet  $(I \cup \{1\} \cup O \cup \{N\})$  and this restriction intersected with an automaton that accepts the language  $[(I \cup \{1\})(O \cup \{N\})]^*$  and it is shown in Figure 7.

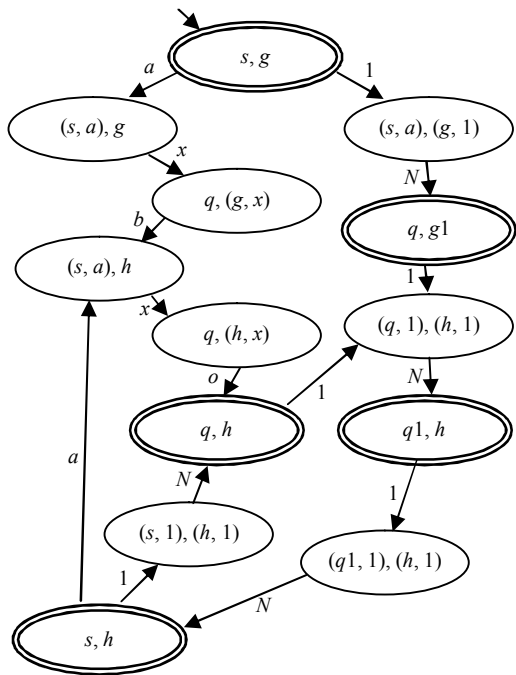


Figure 6 – Intersection of  $Aut(A_S)$  and  $Aut(A_P)$

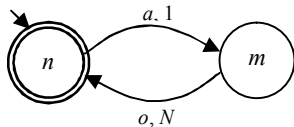


Figure 7 – an automaton accepting language  $[\{a, 1\}\{o, N\}]^*$

We then derive a corresponding FSM coupling inputs and the following outputs (Figure 8) and transform this FSM to a corresponding TFSM (Figure 9) that is the parallel composition of TFSMs  $S$  and  $P$ .

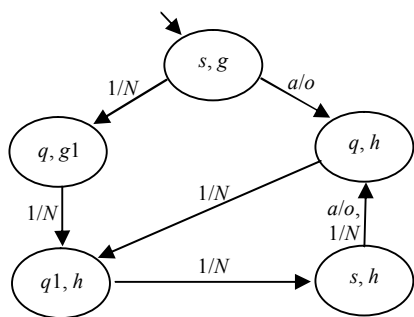


Figure 8 – Composition of  $S$  and  $P$  (FSM)

The state  $(q1, h)$  is copy of the states  $(q, h)$  and  $(q, g1)$ , so there is a time-out equals 2 in the states  $(q, h)$  and  $(q, g1)$ .

Furthermore, the states  $(q, h)$  and  $(q, g1)$  are  $(f)$ -equivalent likewise the states  $(s, g)$  and  $(s, h)$ . That is why we keep only two states in TFSM, shown in Figure 9.

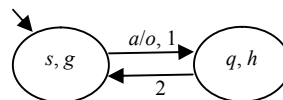


Figure 9 – Composition of  $S$  and  $P$  (TFSM)

#### IV. CONCLUSION AND FUTURE WORK

The propositions 1 and 2 with corollaries give an approach for solving different problems of TFSMs: first, the corresponding FSMs should be constructed, then appropriate methods of FSM theory can be applied to solve the problem of interest and, finally, the result should be converted back to a TFSM. In this paper we used this approach to define, but more importantly, to construct parallel composition of given TFSMs. However, there is a weak point in the presented work. We have not given a proof of the fact, that such a way to construct parallel composition gives a TFSM which describes a system, combined from two TFSMs, operating in the slow environment setting, as it is done for FSM parallel composition [7]. But Propositions 1 and 2 give confidence, that such a proof can be obtained.

Another direction of research with proposed approach, which we want to designate, is solving the TFSM equations. This line of researches is not covered enough in works on timed finite state machines and we believe that known methods for solving the FSM equations can be adapted to TFSMs easily enough.

#### REFERENCES

- [1] М. Громов, Н. Евтушенко, “Синтез различающих экспериментов для временных автоматов,” Программирование, № 4, Москва: МАИК, 2010, с. 1–11.
- [2] M. Gromov, K. El-Fakih, N. Shabaldina and N. Yevtushenko, “Distinguishing non-deterministic timed finite state machines,” in FMOODS/FORTE-2009, LNCS, vol. 5522, Berlin: Springer, pp. 137–151, 2009.
- [3] M. G. Merayo, M. Nunez and I. Rodriguez, “Formal testing from timed finite state machines,” in Computer Networks, vol. 52(2), 2008, pp. 432–460.
- [4] K. El-Fakih, N. Yevtushenko and H. Fouchal, “Testing finite state machines with guaranteed fault coverage,” in TESTCOM/FATES-2009, LNCS vol. 5826, Berlin: Springer, pp. 66–80, 2009.
- [5] A. V. Aho and J. D. Ulman, “The theory of parsing, translation and compiling: Parsing,” New Jersey: Prentice-Hall, 1002 p., 1973.
- [6] A. Gill, “Introduction to the theory of finite-state machines,” New-York: McGraw-Hill, 207 p., 1962.
- [7] Спицына Н. В. Синтез тестов для проверки взаимодействия дискретных управляющих систем методами теории автоматов: Диссертация на соискание ученой степени канд. технических наук. – Томск, 2005. – 158 с.
- [8] М. Жигулин, Н. Евтушенко, И. Дмитриев, “Синтез тестов с гарантированной полнотой для временных автоматов,” Известия Томского политехнического университета, Т. 316, № 5, Томск: Издательство ТПУ, с. 104–110, 2010.
- [9] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence,” Software-Concepts and Tools, vol. 17(3), pp. 103–120, 1996

# Separating Non-Deterministic Finite State Machines with Time-Outs

Rustam Galimullin, Natalia Shabaldina

Radiophysics department

Tomsk State University

Tomsk, Russia

[nihilkhaos@gmail.com](mailto:nihilkhaos@gmail.com), [NataliaMailBox@mail.ru](mailto:NataliaMailBox@mail.ru)

**Abstract**— In this paper we consider one of the classical finite state machine (FSM) model modifications - FSM with time-outs (or timed FSM). In this model in addition to the ordinary transitions under inputs there are transitions under time-outs when no input is applying. The behavior of many modern systems includes time-outs, for example, mobile phones, etc. In the past few years some work have been carried out on studying different relations between timed FSMs. Non-separability relation is very attractive for non-deterministic classical FSMs and FSMs with time-outs course for this relation we don't need «all weather conditions» while testing. In this paper we present and compare two approaches for building a separating sequence for two separable FSMs with time-outs. One of them is using a conversion to classical FSMs, while another one is dealing directly with timed FSMs.

**Keywords** - finite state mashines with time-outs, non-deterministic finite state machines, non-separability relation, timed input sequence, separating sequence

## I. INTRODUCTION

Most of the modern discrete systems, such as digital circuits, telecommunication protocols, logical games, etc., can be described as Finite State Machines (FSM). The entry of FSM receives one of the enabled inputs and returns an output. On condition it is necessary to take into account time aspects of discrete system, time function is interposed[1-4]. FSMs with introduced time function are called FSMs with time-outs or timed FSM (TFSM). Provided that input is being handled uniquely, TFSM is named **deterministic**, otherwise – **non-deterministic**. To distinguish correct and invalid TFSMs distinguishing sequences are generated. They claim exhaustive search of all TFSM's reactions to the input sequence, i.e. it is necessary to input every sequence from test suite enough times to observe all outputs of the system. Practical implementation of this assumption is almost impossible, and it's mostly used to check non-separability relation[5,6]. FSMs are separable[5], if there is an input sequence (called separating sequence), such that the sets of output sequences to this sequence doesn't intersect. In this paper two different approaches for building separating sequence for FSMs with time-outs are suggested.

## II. PRELIMINARIES

Formally, **Finite State Machine** (FSM) is a quintuple  $S = \langle S, I, O, s_0, \lambda_S \rangle$ , where  $S$  is a finite nonempty set of states with initial state  $s_0$ ,  $I$  and  $O$  – finite non-intersecting sets of inputs and outputs,  $\lambda_S \subseteq S \times I \times S \times O$  – transition relation. If for each pair  $(s, i) \in S \times I$  there is at least one pair  $(o, s') \in O \times S$  such that  $(s, i, o, s') \in \lambda_S$ , FSM is called **comlete**. **FSM with time-outs** is a sextuple  $S = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$ , where  $S$  is a finite nonempty set of states with initial state  $s_0$ ,  $I$  and  $O$  – finite non-intersecting sets of inputs and outputs,  $\lambda_S \subseteq S \times I \times S \times O$  – transition relation and  $\Delta_S: S \rightarrow S \times (\mathbf{N} \cup \{\infty\})$  – time-outs function, that defines time-out for every state. The time reset operation resets the value of the TFSM's clock to zero at the execution of the transition. If TFSM, being in certain state  $s_1$ , doesn't receive input for a certain time  $t$  such that  $(s_1, t, s_2) \in \Delta_S$ , it transfers to the state  $s_2$ . FSM is called **observable**, if for each triple  $(s, i, o) \in S \times I \times O$  there is not more, than one state  $s'$  such that  $(s, i, o, s') \in \lambda_S$ . FSM could be considered as FSM with time-outs where for each state  $s \in S$   $\Delta_S(s) = (s, \infty)$ . **Timed input** is a pair  $\langle i, t \rangle \in I \times \mathbf{Z}$ .

Similar to [3], in order to extend transition relation to timed inputs we add a function  $time_S: S \times \mathbf{Z}_0^+ \rightarrow S$  that allows to determine TFSM's state when the clock value is equal to  $t$  based on the current state. Let's consider the sequence of time-outs  $\Delta_S(s) = (s_1, T_1)$ ,  $\Delta_S(s_1) = (s_2, T_2)$ , ...,  $\Delta_S(s_{p-1}) = (s_p, T_p)$  such that  $T_1 + T_2 + \dots + T_{p-1} \leq t$ , but  $T_1 + T_2 + \dots + T_p > t$ . In this case  $time_S(s, t) = s_{p-1}$ . If  $\Delta_S(s) = (s, \infty)$ , then  $time_S(s, t) = s$  for each  $t$ . For each timed input  $\langle i, t \rangle$  we add a transition  $(s, \langle i, t \rangle, s', o)$  to  $\lambda_S$ , if and only if  $(time_S(s, t), i, s', o) \in \lambda_S$ .

Sequence of timed inputs is called **timed input sequence**. Pair  $\alpha/\beta$ , where  $\alpha = \langle i_1, t_1 \rangle, \dots, \langle i_k, t_k \rangle$ ,  $\beta = o_1, \dots, o_k$  is called **timed input/output (I/O) sequence (or timed trace)**, if  $\lambda_S$  defines a sequence of transitions  $(s_0, \langle i_1, t_1 \rangle, o_1, s_{r1})$ ,  $(s_{r1}, \langle i_2, t_2 \rangle, o_2, s_{r2})$ , ...,  $(s_{r(k-1)}, \langle i_k, t_k \rangle, o_k, s_{rk})$ .

As usual, the TFSM  $S$  is **connected** if for each state  $s$  there exists a timed trace that can take the machine from the initial state to state  $s$ .

State  $s'$  is called  $\langle i, t \rangle$ -successor of state  $s$ , if there exists  $o \in O$  such that  $(s, \langle i, t \rangle, s', o) \in \lambda_S$ . The set of all  $\langle i, t \rangle$ -successors of state  $s$  will be denoted by  $\text{suc}_S(s, \langle i, t \rangle)$ , in case of  $t = 0$  we denote it as  $\text{suc}_S(s, i)$ .

An example of TFSM that describes mp3-player behavior is given below:

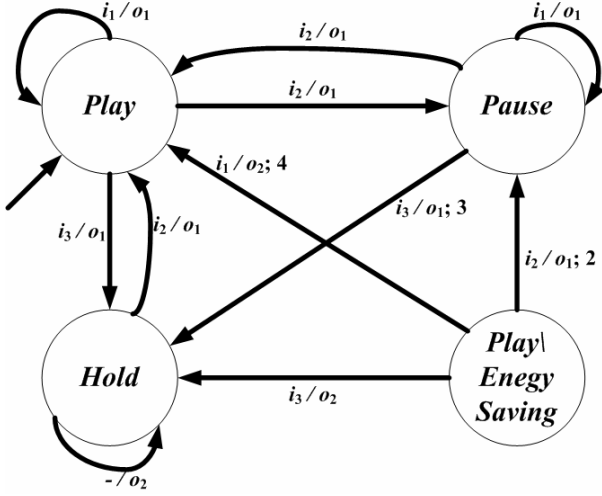


Figure 1. The TFSM that describes mp3-player behavior

The machine consists of the following states:

*Play* – the music is playing, player isn't in energy-saving mode (display's on);

*Play|Energy Saving* – the music is playing, but player is in energy-saving mode (display's off);

*Pause* – the music is stopped, display's on;

*Hold* – the music isn't playing, player's off (hold mode).

Inputs and outputs:

$i_1$  – player's controller is used;

$i_2$  – play/pause button;

$i_3$  – hold button;

$o_1$  – display's on;

$o_2$  – display's off.

Let us observe TFSM's behavior on timed input sequence  $\alpha = \langle i_1, 5 \rangle \langle i_2, 3 \rangle \langle i_1, 4 \rangle$ . In that case the output sequence is  $\beta = o_2 o_1 o_2$ .

A TFSM  $S = \langle S, I, O, s_0, \lambda_S \rangle$  is a **submachine** of TFSM  $P = \langle P, I, O, p_0, \lambda_P \rangle$  if  $S \subseteq P$ ,  $s_0 = p_0$  and each timed transition  $(s, \langle i, t \rangle, o, s')$  of  $S$  is a timed transition of  $P$ .

### III. INTERSECTION OF TWO TIMED FSMs

**Intersection  $S \cap P$  of two TFSMs**  $S = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$  and  $P = \langle P, I, O, p_0, \lambda_P, \Delta_P \rangle$  is the most connected sub-TFSM of  $Q = \langle Q, I, O, q_0, \lambda_Q, \Delta_Q \rangle$ , where  $Q = S \times K \times P \times K$ ,  $K = \{0, \dots, k\}$ ,  $k = \min(\max \Delta_S(s) \downarrow_N, \max \Delta_P(p) \downarrow_N)$ , initial state — quadruple  $(s_0, 0, p_0, 0)$ . Transition relation  $\lambda_Q$  and time-outs function  $\Delta_Q$  are defined according to the following rules[3]:

1. Transition relation  $\lambda_Q$  contains quadruple  $[(s, k_1, p, k_2), i, o, (s', 0, p', 0)]$ , if and only if  $(s, i, s', o) \in \lambda_S$  and  $(p, i, p', o) \in \lambda_P$ .

2. Time function is defined as  $\Delta_Q((s, k_1, p, k_2)) = [(s', k'_1, p', k'_2), k]$ ,  $k = \min(\Delta_S(s) \downarrow_N - k_1, \Delta_P(p) \downarrow_N - k_2)$ . State  $(s', k'_1, p', k'_2) = (\Delta_S(s) \downarrow_S, 0, \Delta_P(p) \downarrow_P, 0)$ , if  $\Delta_S(s) \downarrow_N = \infty$  or  $\Delta_P(p) \downarrow_N = \infty$  or  $(\Delta_S(s) \downarrow_N - k_1) = (\Delta_P(p) \downarrow_N - k_2)$ . If  $(\Delta_S(s) \downarrow_N - k_1), (\Delta_P(p) \downarrow_N - k_2) \in \mathbb{Z}_+$  and  $(\Delta_S(s) \downarrow_N - k_1) < (\Delta_P(p) \downarrow_N - k_2)$ , then state  $(s', k'_1, p', k'_2) = (\Delta_S(s) \downarrow_S, 0, p, k_2 + k)$ . If  $(\Delta_S(s) \downarrow_N - k_1), (\Delta_P(p) \downarrow_N - k_2) \in \mathbb{Z}_+$  and  $(\Delta_S(s) \downarrow_N - k_1) > (\Delta_P(p) \downarrow_N - k_2)$ , then state  $(s', k'_1, p', k'_2) = (s, k_1 + k, \Delta_P(p) \downarrow_P, 0)$ .

#### Algorithm 1: Constructing an intersection of two TFSMs

**Input:** TFSMs  $S = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$  and  $P = \langle P, I, O, p_0, \lambda_P, \Delta_P \rangle$

**Output:** TFSM  $Q = \langle Q, I, O, q_0, \lambda_Q, \Delta_Q \rangle$ ,  $Q = S \cap P$

**Step 1:** add initial state  $q_0 = (s_0, 0, p_0, 0)$  into  $Q$ .

**Step 2:** while set of states of TFSM  $Q$  has non-considered states, consider next in turn non-considered state  $q$ , step 3. Otherwise, **End**.

**Step 3:** for each input  $i$  find state  $q' = (s', 0, p', 0)$  that is  $i$ -successor of state  $q$ . If the set  $Q$  doesn't include  $q'$  – add  $q'$  into the set  $Q$ , add transition  $[(s, k_1, p, k_2), i, o, (s', 0, p', 0)]$  into  $\lambda_Q$  if  $(s, i, s', o) \in \lambda_S$  and  $(p, i, p', o) \in \lambda_P$ .

**Step 4:** if there is a finite delay for state  $q = (s, k_1, p, k_2)$ , then:

$$k := \min(\Delta_S(s) \downarrow_N - k_1, \Delta_P(p) \downarrow_N - k_2)$$

If  $(\Delta_S(s) \downarrow_{N \cup \{\infty\}} = \infty$  or  $\Delta_P(p) \downarrow_{N \cup \{\infty\}} = \infty$  or  $(\Delta_S(s) \downarrow_N - k_1) = (\Delta_P(p) \downarrow_N - k_2)$ ), then  $q' = (\Delta_S(s), 0, \Delta_P(p), 0)$ ;

Else

if  $(\Delta_S(s) \downarrow_N - k_1) < (\Delta_P(p) \downarrow_N - k_2)$ ,  
then  $q' := (\Delta_S(s) \downarrow_S, 0, p, k_2 + k)$ ;

if  $(\Delta_S(s) \downarrow_N - k_1) > (\Delta_P(p) \downarrow_N - k_2)$ ,  
then  $q' := (s, k_1 + k, \Delta_P(p) \downarrow_P, 0)$ ;

Extend function  $\Delta_Q: \Delta_Q(q) = (q', k)$ ;

If the set  $Q$  doesn't include  $q'$ , then add  $q'$  into  $Q$ . **Step 2.**

The intersection of two TFSMs  $S$  and  $P$  (Figures 2,3) is presented in Figure 4.

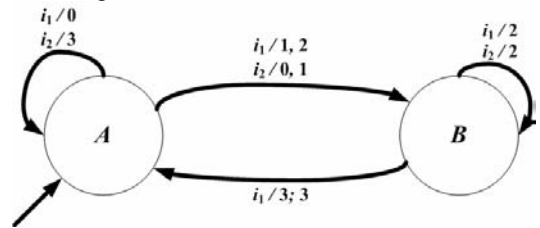


Figure 2. TFSM S

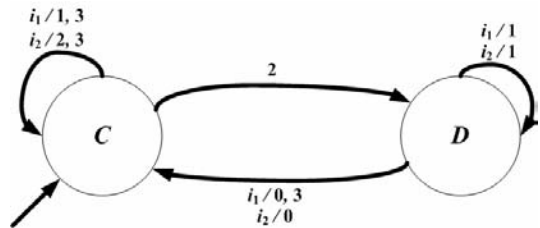


Figure 3. TFSM P

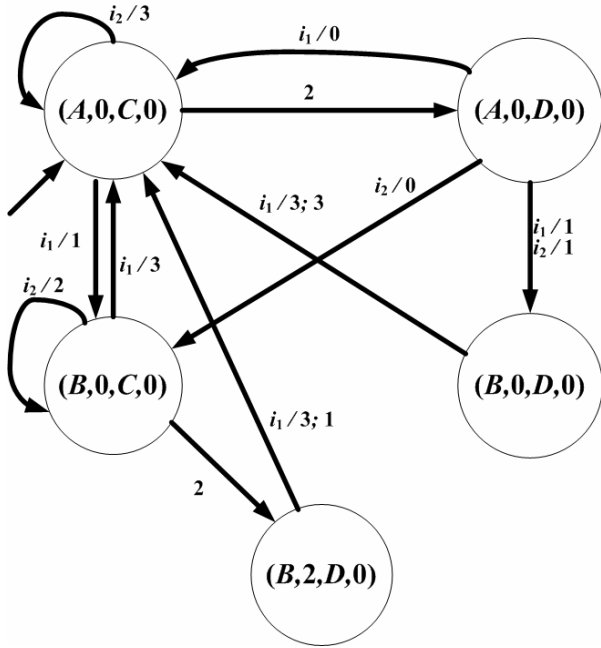


Figure 4. TFSM  $S \cap P$

#### IV. SEPARATING SEQUENCE FOR TWO TFSMS

We suggest algorithm for constructing a separating sequence for two TFSMs.

**Algorithm 2: Constructing a separating sequence of two TFSMs**

**Input:** Complete observable TFSMs  $S = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$  and  $P = \langle P, I, O, p_0, \lambda_P, \Delta_P \rangle$

**Output:** separating sequence for  $S$  and  $P$  (if exists)

**Step 1:** construct the intersection of  $S$  and  $P$  with the help of Algorithm 1. If TFSM  $S \cap P$  is complete,  $S$  and  $P$  couldn't be separated. **End.**

**Step 2:** Derive a truncated successor tree of  $S \cap P$ . The root of the tree is the pair  $\langle q_0, 0 \rangle$ , other nodes – sets of the pairs  $\langle q, t \rangle$ , where  $q$  is the state of  $S \cap P$ .

$k := 0;$

$Edge := \emptyset;$

$Q_{k0} := \{ \langle q_0, 0 \rangle \};$

$Q_k := \{ Q_{k0} \}.$

Until

(Rule 1: for the set  $Q_{kj} \in Q_k, j \geq 0$ , there is an input  $i\text{-sep}$  such that each state  $q, \langle q, t \rangle \in Q_{kj}$ , has no  $i\text{-sep}$ -successors in TFSM  $S \cap P$

or

Rule 2: for each set  $Q_{kj} \in Q_k$  there exists  $Q_{am} \in Q_a, a < k$ , such that  $Q_{kj} \supseteq Q_{am}$ )

Do:

For each input  $i$  construct the set of successors  $M :=$

$\bigcup_{q \in Q_{kj}} \text{suc}_Q(q_{\downarrow Q}, i) \times \{0\}$ , add  $M$  to  $Q_{k+1}$  and add triple

$(Q_{kj}, i, M)$  to the set  $Edge$ .

If there exists  $q, \langle q, t \rangle \in Q_{kj}$ , such that  $(\Delta_Q(q))_{\downarrow (\mathbb{N} \cup \{\infty\})} = \infty$ , define minimum time-out  $T$  and set of successors  $R$  for the set  $Q_{kj} = \{ \langle q_1, t_1 \rangle, \langle q_2, t_2 \rangle, \dots, \langle q_r, t_r \rangle \}$  as follows:

$T := \min_{1 \leq u \leq r} \{ T_u - t_u \}, T_u = (\Delta_Q(q_u))_{\downarrow (\mathbb{N} \cup \{\infty\})};$

$R := \{ \langle q_1', t_1' \rangle, \langle q_2', t_2' \rangle, \dots, \langle q_r', t_r' \rangle \}, q_u' = \text{time}_Q(q_u, t_u + T)$  and either  $t_u' = 0$  if  $(T_u = \infty \text{ or } T_u = t_u + T)$ , or  $t_u' = t_u + T$  if  $T_u > t_u + T$ .

Add  $R$  to  $Q_{k+1}$  and add triple  $(Q_{kj}, T, R)$  to the set  $Edge$ .

**Step 3:** If the tree was terminated according to the Rule 1, then construct the sequence of edges  $(Q_{00}, g_1, Q_{1j_1}), (Q_{1j_1}, g_2, Q_{2j_2}), \dots, (Q_{(k-1)j_{k-1}}, g_k, Q_{kj_k})$  such that  $(Q_{(l-1)j_{l-1}}, g_l, Q_{lj_l}) \in Edge$  for each  $l \in \{1, \dots, k\}$  and  $g_l \in \{I \cup \mathbb{N}\}$ .

Collect the separating sequence  $\alpha = \langle i_1, t_1 \rangle \dots \langle i_m, t_m \rangle:$

$j := 0;$

$T_j := 0;$

$r := 0;$

While  $(j \leq k)$  execute:

If  $g_j \in I,$

Then  $i_r := g_j, t_r := T_j, r := r+1, T_j := 0;$

Else  $T_j := T_j + g_j;$

$j := j+1;$

$m := r;$

$i_r := i\text{-sep}, t_r := T_j.$

If all branches of the tree were terminated according to the Rule 2, then TFSMs  $S$  and  $P$  are unseparable.

**End.**

The truncated tree for TFSMs  $S$  and  $P$  is presented in Figure 5:

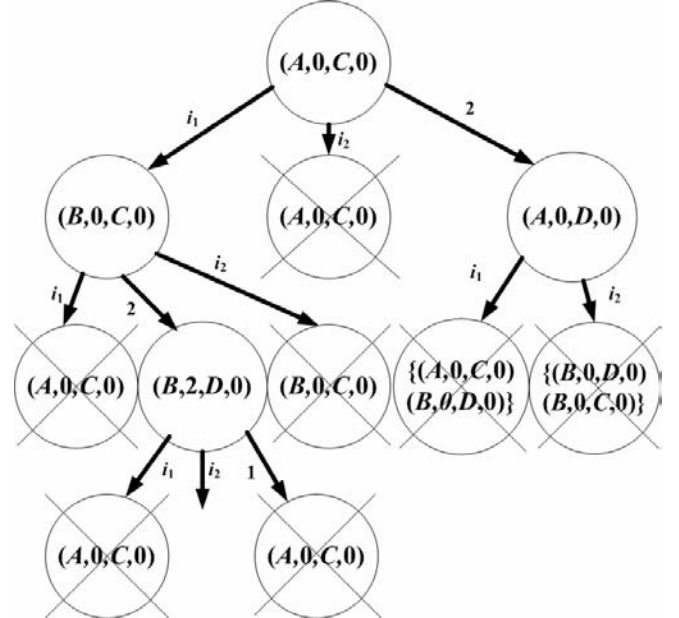


Figure 5. Truncated successor tree of  $S \cap P$

So, the separating sequence is  $\alpha = \langle i_1, 0 \rangle \langle i_2, 2 \rangle$ .

Algorithm 2 is the modification of the algorithm from [6], of deriving a separating sequence for two untimed FSMs. The modifications are associated with time-outs, because the only way to reach some states is to wait for a while. Thus in Algorithm 2 each node of the tree is not the set of states of the intersection, but the set of pairs  $\langle \text{state}, \text{time} \rangle$ . For the set in the node we determine the minimal delay and the set of successors under this delay is derived in the same way as when deriving the intersection. We need the edges labeled by delay because for the timed FSMs the separating sequence is timed input sequence, so we need to wait some time before applying another input.

Rule 2 is inherited from [6] and in this case we can't separate given timed FSMs.

Since Rule 1 is also inherited from algorithm [6], and transitions under time-outs are derived according to the rules that specify the common behavior of timed systems, the sequence  $\alpha = \langle i_1, t_1 \rangle \dots \langle i_m, t_m \rangle$  derived according to Algorithm 2 will be a separating sequence for two timed separable FSMs.

It is known [6], that for given two complete separable untimed FSMs  $S$  and  $P$ ,  $|S| = n$  and  $|P| = m$ , the length of a shortest separating sequence of  $S$  and  $P$  is at most  $2^{nm-1}$ , and this estimation is reachable. Since untimed FSM is a particular case of timed FSM the estimation will be the same.

## V. CORRELATION BETWEEN FSMs AND TFSMs

To transform TFSM  $S = \langle S, I, O, s_0, \lambda_S, \Delta_S \rangle$  into FSM  $A_S$  with the same behavior [3] we add to TFSM a special input  $1 \notin I$  and a special output  $N \notin O$ . FSM  $A_S = \langle S \cup S_t, I \cup \{1\}, O \cup \{N\}, s_0, \lambda_S^\Delta \rangle$  is constructed by adding  $T - 1$  copies of each state  $s \in S$  with finite time delay  $T, T > 1$ . Formally,  $A_S$  is constructed in the following way:

- 1) For each  $s \in S$ ,  $\Delta_S(s) = (s', T), 1 < T < \infty$ , the set  $S_t$  contains each state  $\langle s, t \rangle, t = 1, \dots, T - 1$ .
- 2) For each  $s \in S, \langle s, t \rangle \in S_t$  and for each pair  $i/o, i \in I, o \in O, (\langle s, t \rangle, i, s', o) \in \lambda_S^\Delta$ , if and only if  $(s, i, s', o) \in \lambda_S$ .
- 3) For each  $s \in S$  such that  $\Delta_S(s) = (s, \infty)$  there is a transition  $(s, 1, s, N)$  in  $A_S$ .
- 4) For each  $s \in S$  such that  $\Delta_S(s) = (s', T), T = 1$  there is a transition  $(s, 1, s', N)$  in  $A_S$ .
- 5) For each  $s \in S$  such that  $\Delta_S(s) = (s', T), 1 < T < \infty$ , there are transitions  $(s, 1, \langle s, 1 \rangle, N), (\langle s, j \rangle, 1, \langle s, j + 1 \rangle, N), j = 1, \dots, T - 2$ , and transition  $(\langle s, T - 1 \rangle, 1, s', N)$  in  $A_S$ .

Hence, on condition, that timed FSM  $S$  includes  $n$  states and maximal delay is  $T_{\max}$ ,  $A_S$  can include up to  $n \cdot T_{\max}$  states. Be more precise, number of states in  $A_S$  is  $\sum_{s \in S} n(s), n(s) = 1$ , if

$\Delta_S(s) = (s, \infty)$ , and  $n(s) = T$ , if  $\Delta_S(s) = (s', T)$ .

Thus, in order to derive a separating sequence for two TFSMs, we transformed TFSMs into FSMs. Intersection of

two FSMs could be constructed with the help of Algorithm 1 without taking into account time-outs. To construct truncated successor tree we use Algorithm 2 without time-outs [6], and then collect the sequence (if exists) with the help of step 3 (Algorithm 2). One can assure that the separating sequence for  $A_S$  and  $A_P$  (Figures 6 and 7) will be the same, i.e.,  $\alpha = \langle i_1, 0 \rangle \langle i_2, 2 \rangle$ .

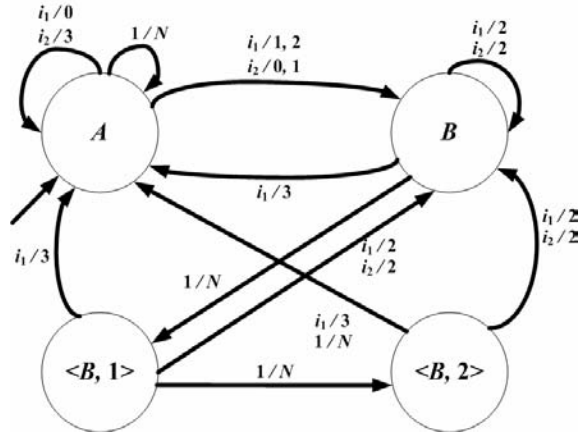


Figure 6. FSM  $A_S$

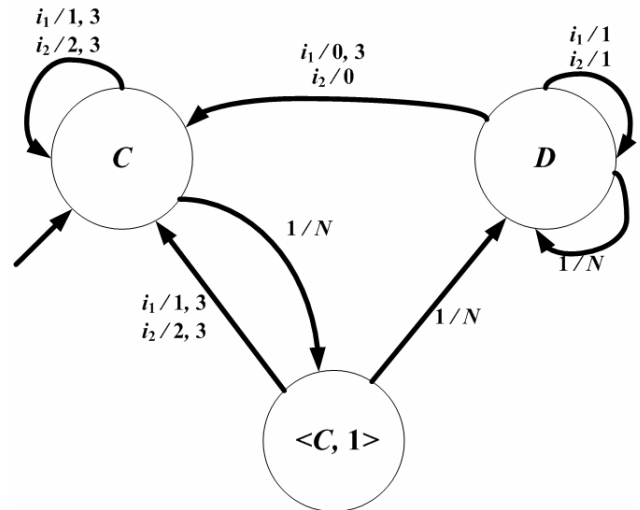


Figure 7. FSM  $A_P$

## CONCLUSIONS

In this paper we suggested two approaches to separate TFSMs. The idea of the first approach is that we construct an intersection of two TFSMs and then find separating sequence. The main advantage of this approach is comparative simplicity due to small amount of states in intersection. Disadvantage – weak theoretical basis of complete test suites derivation for TFSMs. The second approach is based on “TFSM to FSM” transformation. As a result of this transformation we have enormous increasing of states in intersection. Thus this way is hardly applicable to TFSMs with great time delays. But theoretical basis for complete test suites derivation is much more stronger for classical FSMs. In the future we're planning

to compare program implementations of these two approaches in order to find out the range of applicability of each one.

#### REFERENCES

- [1] R. Alur, C. Courcoubetis, M. Yannakakis. Distinguishing tests for nondeterministic and probabilistic machines // STOC'95, NewYork: ACM, 1995. P.363-372.
- [2] M. G. Merayo. Formal Testing from Timed Finite State Machines // Computer Networks. – 2008. – Vol. 52 №2. – P. 432-460.
- [3] M. Zhigulin, S.Maag, A.Cavalli, N.Yevtushenko. FSM-based test derivation strategies for systems with time-outs // Presented to QSIC'2011.
- [4] M. Gromov, D. Popov, N. Yevtushenko. Deriving test suites for timed Finite State Machines // Proceedings of IEEE East-West Design & Test Symposium 08, Kharkov: SPD FL Stepanov V.V., 2008. P.339-343.
- [5] Starke, P.: Abstract automata, American Elsevier, 3–419 (1972).
- [6] N. Spitsyna, K. El-Fakih, N. Yevtushenko Studying the Separability Relation between Finite State Machines // Software Testing, Verification and Reliability. –2007. – Vol. 17(4). – P. 227-241.



# Model Based Conformance Testing for Extensible Internet Protocols

Nikolay Pakulin  
ISP RAS  
npak@ispras.ru

Anastasia Tugaenko  
ISP RAS  
tugaeko@ispras.ru

**Abstract**—Many contemporary Internet protocols are extensible. Extensions may introduce new functionality, alter the format of protocol messages, affect basic functionality or even modify the protocol *modus operandi*. Model based testing of extensible protocols faces a number of problems, the most challenging one is that extensions altering basic functionality require changes in the protocol model. It is highly desirable to have a flexible protocol's model which would let test developers to extend the model without rewriting existing parts of the model. The article presents the method for model based conformance testing for extensible Internet protocols which satisfies this requirement. Each extension is specified in a separate unit, the method presents facility to combine those units into entire model for testing the protocol's implementations. The method uses Java language to formalize the requirements. As an example of the method application article presents test suite development for a number of SMTP extensions.

**Index Terms**—model based testing, conformance testing, extensible protocols.

## I. INTRODUCTION

A lot of protocols for different applications are functioning in the contemporary Internet. Many protocols were developed more than a decade ago. Often while developing a protocol specification it is not easy to foresee all variants of protocol's usage in the future. Protocol application reveals new tasks, makes new demands for the utilizing protocol. The simplest way to update the protocol for new tasks is to develop and publish a new version of the protocol's standard. However for popular protocols this solution may result in necessity of frequent publications of standards' revisions. The high frequency of protocol updates requires much effort for redaction and agreement of the standard's text; it is fraught with injection of indeliberate mistakes and may violate the protocol stability. Also it may hamper the developers of protocols implementations.

To solve the problem of frequent renewal of the protocol specifications nowadays protocol designers follow the pattern that one might call as "extensible protocol". The pattern implies that new protocol features emerging after publication of the protocol standard are specified in separate documents as "extensions". That is, the description structure of extensible protocols consists of the following parts:

- the basic protocol functionality which must be supported by all implementations is specified in a fixed number of

RFC<sup>1</sup> documents, and

- new functions (extensions) are specified in separate RFC documents. For consistent work of extensions with the basic standards developers publish the special RFCs describing general extensions methods.

The history of SMTP protocol [1], [2], [3] illustrates this pattern. Since its inception in 1982 the SMTP protocol went very well with its tasks on sending messages between hosts in computer networks. But the protocol restrictions became apparent in 1990-ies. Then SMTP developers decided to improve some SMTP protocol functions instead of replacing the standard with a new one. It was considered to keep the basic SMTP specifications "as is" and specify new functions in new RFCs. To accomplish this goal in 1995 the RFC 1869 "SMTP Service Extensions" [4] was published. The document specified the method for extending SMTP capabilities.

The SMTP extension mechanism proved to be successful, and the SMTP community agreed to integrate it into the main SMTP specification. In 2001 the revision RFC 2821 [2] was published, it includes the specification of basic functionality and also the specification of extensions mechanisms. Since then all new features, even the fundamental ones – security and authentication, are published in separate documents, leaving SMTP specification intact. The SMTP specification in effect, RFC 5321 [3], contains editorial and clarification changes to RFC 2821 and maintains the extensible architecture of the protocol.

In general protocol extensions may be classified as follows:

- extensions specifying new functionality (e.g., new commands, new types of messages and responses);
- extensions altering the format of protocol messages;
- extensions altering the basic protocol functionality or operations of other extensions;
- extensions altering *modus operandi* of the protocol.

The classification of the extensions provided above shows, that extensions shouldn't be considered as independent – they may affect each other or the basic protocol. This observation leads to a conclusion that a model of an extensible protocol is not a plain composition of the basic protocol model and models of extensions. Modeling of extensible protocols requires a specialized composition that takes into consideration

<sup>1</sup>RFC – Request for Comments – the normative document for the Internet standards

dependencies between extensions.

Another important aspect of protocol extensibility is optional support of extensions in various implementations. Implementors are not obliged to support all published extensions; as a result, implementations of the same protocol might provide different sets of protocol extensions. This observation and the fact that extensions may change the basic protocol's functionality hampers testing of extensible protocols. To test extensible protocols test developer must know the exact set of extensions supported by Implementation Under Test (IUT, also we will call it "target implementation") and take into account the profile of those extensions to assign the right verdict concerning IUT conformance to the protocol specification.

As a consequence, model based testing (MBT) of extensible protocols faces a number of problems. First of all, the protocol model must be coherent with the IUT, i.e. the model must reflect the extensions, provided by the IUT, and leave all other extensions beyond the scope. Furthermore, the model must take into account that extensions may alter basic protocol functionality. Obviously, it is unwise to develop independent models for each thinkable set of protocol extensions. A more realistic scenario is to construct models of the protocol and its extensions separately and to combine them into the model of IUT depending on the actual extensions profile of that IUT. To implement this scenario the test development method should provide facilities for modular specification development, and facilities to combine the specification modules either statically or dynamically to match the list of supported extensions of arbitrary IUT. Despite the challenge with modular specification of extensible protocol it is highly desirable to have modular test specification, when test actions for extensions are specified in separate units and the actual test is composed from test specifications of extensions provided in IUT.

The article presents the method for model based conformance testing for extensible Internet protocols. Within the proposed method basic protocol functions and extensions are modeled as state machines with common state. Each state machine is specified in a separate unit, the whole model of IUT is composed from basic protocol specification and extensions before test execution starts. Authors has developed a composition facility that makes possible modular specification of extensions that introduce new functions, alter protocol specification and, to some extent, extensions that modify protocol message format. The proposed method does not allow specification of extensions that change modus operandi of a protocol (such as PIPELINING extension of SMTP [5]). The method supports modular development of test actions. Test is treated as a finite state machine and the method implies specification of test transitions for extensions separately. The method provides a facility that combines test specifications of different extensions into a single test state machine dynamically before test execution.

The paper is structures as follows. Section II presents existing approaches to model-based conformance testing and discusses their applicability to testing of extensible protocols. Section III describes our previous work on model based testing

of electronic mail protocols. Section IV presents a new model based method for testing protocols with extensions. Section V consists of two parts, the first part describes SMTP protocol and a few extensions, and the second part presents an example of new method application – the development of a test suite prototype for SMTP protocol with extensions. Section VI discusses the applicability of a presented method. Section VII presents results of the work and describes directions for future work. And Section VIII is a conclusion.

## II. RELATED WORKS

Internet protocols contain multitude of states, those states may be divided into groups with similar behavior inside each group. Manual test suite development for such kind of protocols seems very laborious and redundant because of similar checks in similar states. Tools for automated testing which allow re-use of a protocol model for verdict assignment become more frequent in use for developing test suites for Internet protocols.

In general, it is convenient to use tools possessing the following properties for testing Internet protocols:

- formal relation between requirements and tests;
- automated verdict assignment about the correctness of IUT behavior.
- automated tests generation depending on IUT responses;

Testing of protocol's extensions brings in supplementary requirements:

- the ability to easily change a protocol's model. Some extensions add new commands, new response codes, new states thereby changing the protocol's model. To develop test suites for protocols extensions test developer must have an opportunity to choose the right model: either basic or modified by a number of supported extensions;
- the ability to develop specifications and tests as separate units, one unit per extension, basic specification should be specified in separate units. This requirement is dictated by the fact that extensions are optional and not all implementations must support all extensions.

The majority of listed above requirements are satisfied by application of the model based approach and tools.

We don't consider the JUnit [7] and TTCN-3 [8] as they are not model based, the test suites written with these tools contain a lot of redundant code, test sequences must be constructed manually and all the verifications also must be specified manually. The whole process of test suite development for extensible protocols with these tools is very laborious because of a greater part of the basic test suite (test suite for testing the basic protocol's functionality) must be changed.

The UPPAAL [9] is a popular tool suite for model checking of real-time systems. It provides checking of models with hundreds of states, has some facilities to detect deadlocks. But testing of Internet protocols deals with black-box state machines, so test developers need a tool for automated test suites generator, and the UPPAAL doesn't provide such generator.

Tools NModel [10] and SpecExplorer [11] may be successfully used for developing test suites for simple small-scale protocols but they require development of considerable supplementary libraries to test large-scale protocols and extensions. Toolkit Conformiq Qtronic [12] doesn't provide tests generation, as a result it generates TTCN-3 scripts.

### III. PREVIOUS WORK

In works [13], [14] the method for automated testing of e-mail protocols was presented. That method uses Java specification extension JavaTESK [15]. We used that method to develop test suites for basic functionality of implementations of mail protocols SMTP, POP3 and IMAP4 [14]. Developed test suites were applied to implementations and detected a number of noncompliances in well-known open-source mail servers.

But that method isn't suited for conformance testing of extensible protocols since it does not support modular models and tests and implied adaptation of the protocol model and test for each extension.

### IV. METHOD FOR EXTENSIBLE PROTOCOLS TESTING

This section presents a method for model-based conformance testing of extensible Internet protocols' implementations. The primary target of the method is to provide test developers facilities for modular modeling and test specification. As stated in the introduction, protocol extensions are not independent. On the contrary, protocol extensions may affect each other; furthermore, extensions may even alter the basic functionality of the protocol. These observations lead to a conclusion, that a model of an extensible protocol can not be presented as a plain composition of separate models.

The method consists of two parts:

- 1) All the extensions are specified in separate units, in general one unit represents one extension. The structure of the model state is specified in the separate classes and all test units use this global shared state. Every unit may change the current state according to extension's specification. Basic functionality is also specified in one or several units (e.g. connect and disconnect may be specified in one unit and transaction commands – in the different unit). To model implementations with extensions the structure of general shared state may be changed by the following actions: addition of new symbols of the states introduced by the extensions; specifying new actions; specifying or changing precondition for actions from different states.
- 2) All test state machines for extensions are specified separately in different units. Whole test is constructed as a composition of test state machines for extensions supported by IUT. Note that this is not a true "composition" because of general test utilizes the shared state. Abstract test description for each extension is specified in a separate unit.

Application of this method defines the following actions to test extensible protocols. For adding new extension test

developers add specification and test units into the project. If an extension adds new command then this command should be added to the commands (actions) register. Also the availability of this command in the set of states should be added to the states register. If extension changes the basic functionality then test developers should rebind old aspect with new specification unit. The comprehensive test is constructed as a composition of units specifying the extensions supported by the IUT.

The proposed method contains the following main aspects:

- simplification of the development and maintenance of model and tests for protocol's implementations. Specifications and tests for the protocol's extensions are defined in separated modules. This allows easily changing the protocol models for new protocol's extensions;
- construction of specification for IUT in accordance with set of extensions supported by target implementation;
- construction of general test as a composition of test modules for extensions supported by the target implementation.

The method utilizes the library for model based automated testing [16]. The protocol's model presented as a finite state machine. The implementation's requirements presented in contract specification notation: for each operation pre and postconditions are defined, precondition restricts the operation's availability in different states, postcondition specifies the required behavior. Also the library grants the following useful tools:

- test sequence iterator. The test sequence generates automatically from test model and contract specifications. Test model is presented as a finite state machine, each impact to the IUT contains precondition which specifies the acceptability of this impact in the current state;
- automated coverage calculation. States and transitions may be labeled with marks and branches, in this case they will be represented in the test report. Such labels allow defining the formal relation between requirements and tests;
- automated verdict assignment concerning the implementation under test behavior.

The presented method contains the following steps:

- 1) Creation of requirements catalogue for basic specification of the protocol. This catalogue contains only requirements from basic protocol specification and doesn't include the requirements from extensions'.
- 2) Creation of requirements catalogues for protocol extensions. These catalogues contain requirements from extensions specifications.
- 3) Designing of the extensible model for the basic specification. The extensibility of the model will be necessarily in the next steps;
- 4) Designing of units for extensions' specification. Generally, one unit represents one extension. These units may be easily included into protocol's model developed in the previous step;
- 5) Designing of formal specification for basic protocol and

extensions. The basic functionality specification is developed as self-sufficient. The extensions' specifications are developed as separate units which may be added into the basic specification.

- 6) Formalization of requirements. At this step the requirements of basic specification and the requirements of extensions are formalized as pre and postconditions.
- 7) Developing of test scenarios for basic functionality.
- 8) Developing of units with test scenarios for protocol extensions.
- 9) Constructing of the comprehensive test for the target implementation. This test includes the scenarios for basic functionality and the scenarios for extensions supported by implementation under test.
- 10) Execution of test suites and analyzing the results. Test suite improvement when necessary.

Not all steps in the method are mandatory. Also note that this method may be used not only for extensible protocols and for extensions, it may be easily utilized for testing other kinds of Internet protocols. If in the last step not all requirements are covered by the constructed test than steps 6-10 may be repeated as many times as needed.

## V. METHOD CASE STUDY

This section presents the method case study on testing the extensible Simple Mail Transfer Protocol (SMTP). First subsection contains the description of SMTP protocol and its' extensions; second subsection presents the description of test suite development.

### A. SMTP Protocol Extensions

Protocol SMTP is a text based protocol of the upper layer of the TCP/IP stack. The protocol consists of two parties: a client and a server. After establishing a connection the client issues commands to the server and the server executes them and returns responses to the client. The response depends on success of the command execution.

Simple mail transfer protocol is used to send messages. This protocol has a following feature: each physical server could operate as both SMTP server and SMTP client. Being a server it accepts incoming emails and then became a client to forward these received messages to the next hops. To forward messages between various domains SMTP uses its own overlay network over TCP/IP. When an SMTP implementation being a server identifies itself as the final destination of the message it stops forwarding the message and places it into internal implementation-specific storage. To retrieve emails from the storage end-users utilize other protocols: POP3 (Post-Office Protocol, version 3 [17]) or IMAP4 (Internet Mail Access Protocol version 4 [18]).

SMTP protocol is extensible. To identify what extensions are supported by a server implementation a client should issue the EHLO command. To this command the server replies with multiline response, lines provides information about supported extensions. Response lines include extension-specific keyword and also may contain any supplementary information.

The basic protocol model consists of the following states: *DISCONNECTED*, *CONNECTED*, *AFTER\_HELLO* (after issuing EHLO or HELO commands), *AFTER\_MAIL\_FROM* (after issuing MAIL command), *AFTER\_RCPT\_TO* (after issuing RCPT command), *AFTER\_DATA* (after issuing DATA command), *AFTER\_DOT* (after issuing the sequence  $\langle CRLF \rangle . \langle CRLF \rangle$  in the *AFTER\_DATA* state). With respect to specification states *AFTER\_EHLO* and *AFTER\_DOT* are the same, we divide them in model to test implementations to conform to this requirement.

Lets consider a few examples of SMTP extensions and categorize them according to extensions classification proposed in Introduction I. The DSN extension described in RFC 3461 [19] specifies the Delivery Status Notifications (DSNs). For example, the client may specify that DSN should be generated under certain conditions (e.g. when the mail has reached the recipient) and sent to the initial client. To use this option initial client should add new parameters in the transaction commands. This extension is of type 1 – bringing in new functionality.

The AUTH extension specified in RFC 4954 [20] adding new state to the protocol model. If an implementation supports this extension then the basic set of commands would not be enough to send a message: server may require client's authentication. This extension also introduces new command AUTH, new parameters for MAIL command and new response codes. For example, a server may response to the transaction commands with new code 530. In this case such response means that authentication is required. The STARTTLS extension described in RFC 3207 [21] specifies the TLS usage which helps SMTP agents to protect all or few interactions from interceptions and attacks. This extension also adds new state into the protocol model, specifies new command STARTTLS and new response codes. If a server supports this extension the transaction commands are not allowed before the command STARTTLS. These extensions are of type 3 – altering the basic protocol's functionality.

The PIPELINING extension specified in RFC 2920 [5] provides a facility to group several commands to send them in one transfer operation. If a server supports this extension it may response to the group of commands as a whole instead of sending responses to separate commands. This extension changes the protocol's structure and is of type 4 – altering the protocol's modus operandi.

Protocol SMTP has a long history. Nowadays the basic specification [3] includes the mechanisms of extensions and provides different optional parameters in basic commands. So protocol SMTP has no extensions of type 2 (extensions altering the format of protocol's messages), all extensions which provides new parameters are fitted into the extensible messages format.

### B. Test Suite Development for SMTP Protocol Implementations with Extensions

The authors have developed a prototype of test suite for SMTP protocol using the presented method. For basic specifi-

cation we used the requirements catalogue from the previous works [14]. Also we made new requirements catalogues for two extensions: the AUTH extension [20] and the DSN [19] extension which cover two types of extensions (new functionality and altering basic functionality). For these extensions we develop new separate units with specifications (one unit for one extension) and new separate units with tests (one unit for one extension as well) .

The structure of the protocol model is organized as follows. We have a set of states and a set of actions. We define two maps from states to actions (we name them **allowed** and **denied**) which define policy which commands are allowed or denied in particular states. If an extension adds a new command we add this command to the actions set and define allowed/denied policy for this action. If an extension adds a new state we add this state to the states set and extend allowed/denied policy of the protocol commands for this state. Note, the pair state-action may be undefined in both **allowed** and **denied** maps. In this case we can provide two types of testing: the conformance testing, in which we consider undefined pair as denied; and the robustness testing when undefined pair is considered as allowed. In the latter case we try to send the command from pair in the state from this pair and looks whether the target implementation is down.

For testing the AUTH extension we defined a new state AFTER\_AUTH and updated the allowed/denied policies for transaction commands (MAIL FROM, RCPT TO and DATA). If the implementation supports the AUTH extension the transaction commands may be issued only in authorized states. Also we added new command AUTH and the parameter AUTH for MAIL FROM command – the AUTH extension provides two authentications mechanisms. Then we updated maps **allowed** and **denied** to contain the information about allowed and denied transitions.

For testing the DSN extension we used methods for MAIL and RCPT commands with optional parameters. Since this extension adds only new parameters we didn't change the protocol model.

We defined a configuration file with a list of extensions supported by IUT. Then we used tool [16] to construct the whole model of IUT (from units specified above) and generate a test suite. Generated test allow detecting the following types of noncompliances:

- missing required commands;
- protocol rules violation, such as accepting commands in illegal states;
- wrong reply codes to the protocol commands.

## VI. DISCUSSION

Presented method is applicable for synchronous message based protocols. In such protocols clients send commands to the servers and servers executes them and returns the responses to each command. Responses contain the code which defines the success of the command execution.

Protocol model consists of few parts: basic part which represents the model of the basic protocol's specification and

supplementary parts for protocol's extensions. Novelty of this method is the ability to easy altering the protocol's model and adding new tests.

The method was assayed by the development of test suites for SMTP protocol implementations with extensions. The SMTP extensions may add or alter the protocol's basic functionality, bring in new states, new commands and new response codes. The prototype of test suite for testing SMTP implementations with extensions shows the applicability of the method for testing extensible Internet protocols.

## VII. RESULTS AND FUTURE WORK

The particular method for testing extensible Internet protocols is presented. The development is in progress, currently we have a method for testing a few types of extensible protocols. Protocol's extensions which we can test with developed method possess the following characteristics: they may add new commands, new responses, new model states but they must not alter the protocol's structure (modus operandi) and also they must not bring in new encodings of symbols of sending messages.

Using this method we have developed the prototype of test suite for testing SMTP implementations with a number of extensions. The current version of method isn't applicable for all types of extensions. For example, the extension PIPELINING [5] for protocol SMTP changes the structure of the protocol. If this extension is supported implementation from message-based became stream-based and requires other testing methods.

Most Internet protocols possess a command for identifying the list of supported extensions. The current version of presented method utilizes a configuration file to construct the modified model of IUT. In future versions we plan to add a feature of dynamic composing of the model and test state machine depending on implementation responses to the capabilities command.

After the tests has been executed test developers got a test trace. This trace contains the log of test execution, so it contains important information on what is wrong with implementation under test. Separately of this method we have a report generator, generated reports presents the test trace demonstrably but not obviously. Currently test developers should manually find the places in the test which shows the noncompliances with the specification. To operate with obtained information more easily we plan to improve the report generator.

## VIII. CONCLUSION

The paper presents a method for automated model-based conformance testing of implementations of extensible Internet protocols. The modeling approach uses state machines to express functional specifications as a formal definition of textual requirements elicited from normative sources. Test is a traversal of some simplified (compared to the model) state machine; the sequence of test stimulus is generated depending on IUT responses.

The main idea of the method is modular approach to test suite development: both functional specifications (models) and test specifications of basic protocol functionality and each extension are developed in separate units. Models of extensions are expressed as state machines over common extensible set; the method provides facilities to combine such partial models into a complete state machine, depending on the exact set of extensions supported by a specific IUT. Test is constructed as composition of test state machines of the extensions supported by the specific IUT.

The characteristic feature of the proposed method is the choice of notations for models and test specification. We use programming language Java, pure, without any extensions (such as Java Modeling Language, JML [6]), while model composition is partially defined in XML. The selection of Java as the primary notation gives the full power of Java expressiveness, rich toolkits for model and test development and, potentially, has more chances to attract attention of industry since the method does not require experts in formal description languages.

Using this method the prototype of test suite for testing SMTP protocols with some extensions was developed. Test suite covers the following types of protocol's extensibility: adding new functionality and altering the basic protocol's functionality. The extensions which are altering the protocol's modus operandi have not been tested yet. The development of new method is ongoing project and extending this tool to test the extensions altering the protocol's modus operandi is one of the tasks to decide. Also we plan to improve the report generator and to extend the method and tool for testing more types of Internet protocols' extensions.

#### ACKNOWLEDGMENT

The authors would like to thank Victor Kuliainin for kindly provided Java library for automated model based testing.

#### REFERENCES

- [1] IETF RFC 821. Jonathan B. Postel. Simple Mail Transfer Protocol. 1982.
- [2] IETF RFC 2821. J. Klensin. Simple Mail Transfer Protocol. 2001.
- [3] IETF RFC 1869. J. Klensin. Simple Mail Transfer Protocol. 2008.
- [4] IETF RFC 1869. J. Klensin, N. Freed, M. Rose, E. Stefferud, D. Crocker. SMTP Service Extensions. 1995.
- [5] IETF RFC 2920. N. Freed. SMTP Service Extension for Command Pipelining. 2000.
- [6] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, pages 342-363. Volume 4111 of *Lecture Notes in Computer Science*, Springer Verlag, 2006.
- [7] Unit testing framework, <http://www.junit.org>.
- [8] ETSI ES 201 873-1 V3.1.1. Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. Sophia-Antipolis, France: ETSI (2009).
- [9] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal 4.0. <http://www.uppaal.com/>
- [10] Jacky, J., Veanes, M., Campbell, C., Schulte, W.: *Model-based Software Testing and Analysis with C#*. Cambridge University Press, Cambridge (2008).
- [11] <http://research.microsoft.com/pubs/77383/bookChapterOnSE.pdf>  
<http://research.microsoft.com/en-us/projects/specexplorer/>
- [12] End-to-End Testing Automation in TTCN-3 environment using Conformiq Qtronic and Elvior MessageMagic. 2009.
- [13] A. Tugaenko, N. Pakulin. Test suite development for conformance testing of email protocols. // *Proceedings of Spring/Summer Young Researchers' Colloquium on Software Engineering*, pp. 87-91, Nizhny Novgorod (2010).
- [14] N. Pakulin, A. Tugaenko. Specification Based Conformance Testing for Email Protocols. // *Proceedings of ISoLA 2010*, pp.371-382. Heraclion, Greece, 2010.
- [15] JavaTESK: getting started. Moscow, 2008.
- [16] V. Kuliainin. Component architecture of model-based testing environment. *Programming and Computer Software*, 36(5):289-305, 2010.
- [17] IETF RFC 1939. J. Myers, M. Rosem, Post Office Protocol – Version 3. 1996.
- [18] IETF RFC 3501. M. Crispin. Internet Message Access Protocol – version 4rev1. 2003.
- [19] IETF RFC 3461. K. Moore. Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs). 2003.
- [20] IETF RFC 4954. R. Siemborski, A. Melnikov. SMTP Service Extension for Authentication. 2007.
- [21] IETF RFC 3207. P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. 2002.

# Developing test systems for multi-modules hardware designs

Mikhail Chupilko

Institute for System Programming of RAS  
Moscow, Russia  
chupilko@ispras.ru

**Abstract**—The paper proposes the approach of creating test systems for complex hardware designs. The designs can be subdivided into modules and verified separately. The proposed architecture of separated verification systems and the way to combine them into a complex test system are based on simulation-based verification of hardware designs. The components of test systems are connected in a TLM-like way that allows to use high-level model of commutation based on messages and thereby to simplify merging of several test systems into a test system for the complex component.

**Keywords**—complex hardware designs; simulation-based verification; combination of test systems

## I. INTRODUCTION

The importance of *hardware verification* has been urgent for many years. There are several techniques to conduct it but there is still not unified solution. *Hardware designs* are developed by means of languages of hardware behavior description (hardware description languages, HDLs, e.g., Verilog [1]). Even relatively simple *modules* (i.e., parts of complex designs or very simple designs) can hardly be checked by means of a code inspection, to say nothing of complicated designs. Therefore, the verification, i.e. checking of designs' behavior and their specification to mutual consistency, is of importance and attention. There are some estimates talking that about 70% total amount of development efforts are spent on the verification [2]. The real practice shows that usually at least half of total design development time is spent on this aim. The code written in HDLs called *HDL-model* can be translated into a *net-list* and then on this base the real device will be created. If no modifications of a net-list take place, the functionality of the produced device will be the same as of the HDL-model. Even if the corrections took place, the equivalence can be checked by means of special tools, e.g. [3]. Therefore, to reveal and correct functional errors is possible at the stage of HDL-model development. It should be noticed that the correction of functional errors on later stages and, in particular, after chip manufacturing requires more efforts and time, because necessity to pass all stages of manufacturing again.

Complicated designs are usually developed by means of *abstraction* and *decomposition* techniques. The common approach is to develop the whole system abstractly and then to create the subparts (modules) more carefully. Usually the test system for the whole design under verification (DUV) is created but it is rather abstract one. As some parts of the system

can be critical for the total system behavior, the module-oriented test systems are developed. If these “little” test systems can help to improve the common test system for the whole DUV, it will be good for code reuse and debugging abilities of the common test system.

This work is organized as follows. First, the review of works related to verification of complicated hardware designs is given. In the following section, the architecture of test systems formerly proposed in [4] is described. Next section introduces the architecture of multi-module test system. Section 5 includes case studies. Section 6 concludes the paper.

## II. RELATED WORKS

There are two common ways of hardware verification. First, we could use *formal methods*, e.g. to prove satisfiability of a logical constructions in a formal model based on hardware design in model checking [5]. All corrections are made over static hardware designs. They are applicable well in case of module-level verification but their scalability is insufficient [6] to use them in case of the whole designs. To improve the scalability, the hardware designs can be checked dynamically during the *simulation* process. Simulation-based verification allows checking the designs in real cases of their work. Usually, simulation-based approaches possess the high level of scalability and the thoroughness of verification varies according to available resources and time. Below, when we are talking of verification we are meaning only simulation-based verification.

Typical components of the test systems are *test sequence generator (stimuli generator)*, *reaction checker (or oracle)*, *test completeness estimator*. The generation of sequence or stimuli can be made manually and explicitly by means of test cases description. Other stimuli generators produce test actions self-automatically requiring manual description of variables set in each stimulus with restrictions for their values. To generate stimuli, special mechanism selects subset of available stimuli, solves constraints in their variables assigning fit numbers to the fields and start stimuli. This approach is called *constrained-driven verification (CDV)* [7]. Another well-distinguished way of stimuli generation is *FSM traversing* [8], where states of the FSM are states of system under test and transitions between them are applied operations. The reaction checker always knows the correct behavior of DUV, e.g. utilizing a *reference model*. Test completeness estimator usually works on the base

of source code coverage or functional model code or aspects coverage.

The main subject of the article is the possibility for developing test systems allowing reuse in the multi-module complex design case. To reduce extra problems we suggest using a uniform architecture for all test systems. When merging test systems the question about merging of each component from their structure arises. Consequently, the parts should be intended for easy merging initially, their architecture should provide such possibilities.

Among well-spread approaches of verification, we selected *Open Verification Methodology (OVM [9])* as the most spread and seemed to be most suitable for our purposes to observe its abilities of merging. Test systems according to OVM are developed in accordance to the given architecture and subdivision of test system's components into several layers [10]. The test system for each single module is named *Open Verification Component (OVC)* (see Figure 1). Each OVC contains basic means of creating CDV stimuli flow (*transaction sequencers*, where *transaction* is a *abstract message* containing information about test situation) and delivering the flow to the DUV (so called *transactors*, i.e. components which make direct and reverse transformation of transactions and DUV's wire signals). OVC can be connected to each other when they are put under control of the *united test controller* or in other words *virtual generator* [7]. In this case, all the OVCs will generate stimuli flows. Developer of test system modifies the redundant generators connected with unavailable DUV's wires to switch them off. The OVCs with turned off generators check their target modules correctness regarding to the stimuli flow to the components influence their target modules work. Components checking correctness (or *scoreboards*) continue checking using only available data from DUV. Summarizing, test systems made according to OVM satisfy many tasks usually arising in verification including connection of several test systems. It should be noticed that OVM is oriented to programming in SystemVerilog so that connection with other languages is possible but knotted with development of intermediate components.

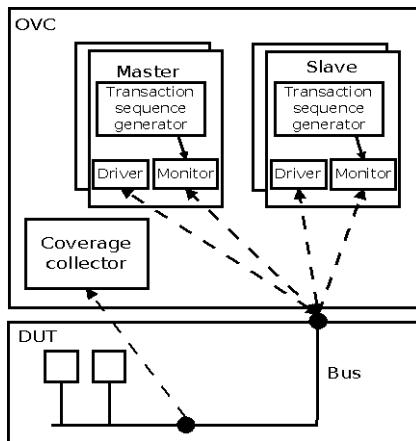


Figure 1. Open verification component

We developed new approach and presented some aspects of it in [4]. That paper touched upon only the problems of oracles' development for single test systems. We will shortly review the

approach in this section and thoroughly in the next. The method utilizes simulation-based verification and implies the subdivision of test system components generally into *stimuli generator* and *oracle* (or *reaction checker*). The generator should create a flow of stimuli and apply them into the oracle usually based on a reference model developed on selected level of abstraction. Test is complete when the generator indicates it according to a strategy of generation.

This approach has a distinctive feature when compared with OVM: the reference models used in reaction checkers can be originally written at a high level of abstraction and then can be specified according to a progress of DUV development. To make it possible, there are several techniques in the approach such as model reactions' arbitration mechanism, DUV reactions' detection mechanism, etc. After all, while DUV is developed, the verification engineers usually develop the *software simulator* of the total DUV. As they usually do it by means of C++, to use exactly this language is useful to reuse parts of simulators to create reference models with no extra efforts. As the approach described in [4] uses C++, it has a certain advantage over OVM while task of system simulator reuse is conducting.

### III. ARCHITECTURE OF SINGLE TEST SYSTEM

The architecture described in [4] was based on UniTESK technology [11] developed in Institute for System Programming of RAS. The architecture includes *stimuli generators* (including *FSM-based one*), *oracle* (*reaction checker*), and after all *coverage tracker* and *verification report generator* (see Figure 2). The stimuli generator produces sequence of *messages* and sends them as a parameter while calling interface operations of the reference model. These calls are named sending *model stimuli*:

```

dut.start(&DUT::pop_stimulus, dut.iface1, msg);

```

Reaction checker processes messages and makes *model reactions*. Then it sends stimuli (which now are called *design stimuli*) to the *target design* and receives its *design reactions*. At last, reaction checker checks correspondence between model and design reactions and returns a *verdict* about correctness of DUV at the current cycle. The coverage tracker saves the information about *functional coverage* at the current cycle. The report generator dumps important information about the verification process such as called operations, reached functional coverage and verification result.

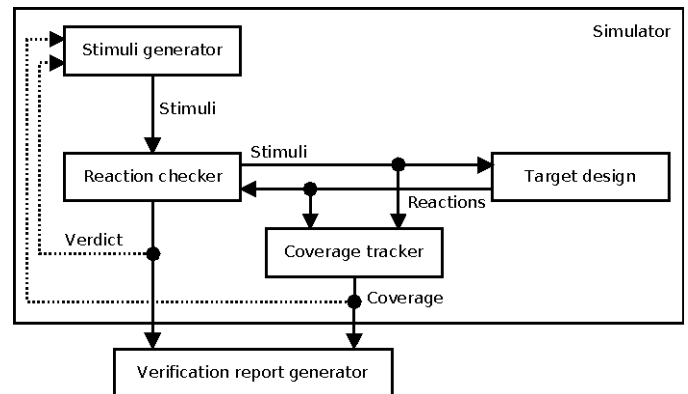
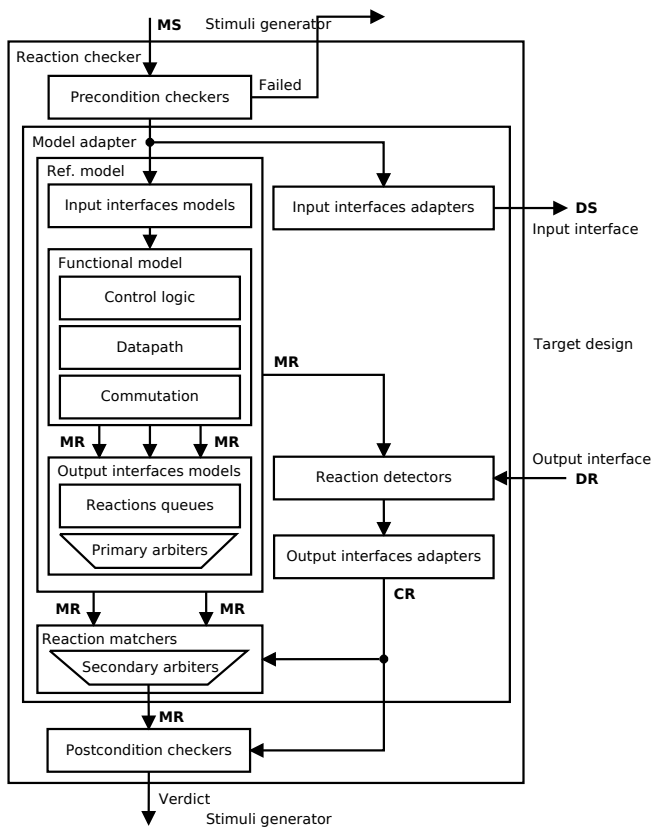


Figure 2. Single test system architecture



The most complex component requiring and allowing reuse is reaction checker (see Figure 3). The reaction checker supplies the reference model with all the necessary functions, which make it possible for the stimuli generator (or other reaction checker) to utilize the reference model and model adapter.



MS - Model stimulus (abstract message)  
DS - Design stimulus (cycle- and pin-accurate serialization of MS)  
MR - Model reaction (reference message or constraint)  
DR - Design reaction (cycle- and pin-accurate series)  
CR - Checked reaction (deserialization of DR)

Figure 3. Reaction checker architecture

The messages sent into the checker are called *model stimuli* (**MS**). The generator (or other reaction checker) addresses the **MS** flow to one of the *input interfaces models*.

On having received the **MS**, *pre-condition checkers* check if the **MS** can be started. The **MS**, which starting requirements are not satisfied at the current state of the functional model, is rejected. In the other case they proceed both to the DUV via *input interface adapters* (in this step processed **MS** is called *design stimuli*, **DS**) and to the *functional model* via *input interface models* set by the generator.

The functional model produces *model reactions* (**MR**) and places them into one of the *output interface models* according to rules included into the functional model.

The output interface models contain *reaction queues* keeping **MR** and *primary arbiters* selecting **MR** subset at the current simulation cycle. The arbiters work according to a strategy selected by the test developer. The **MR** subset is sent

into reaction detectors to help recognizing *DUV's reactions* (**DR**).

The *reaction matchers* fetch the **MR** sub-subset from the output interfaces models and then start expecting the corresponding reactions from the **DUV**. The restrictions are made by the second arbiters and customized by test developer. There are certain time restrictions for the waiting. If they are violated, the test system shows the *timeout error* and stops working.

When the **DR** are found, they are put into one of the *output interface adapters* (corresponding **MR**, if it is found, helps to select which one). If some of **DR** does not have the corresponding **MR**, the test system shows an *unexpected reaction error* and stops working.

The output interfaces adapters send the *checked reactions* (**CR**) into the reaction matcher to find the corresponding **MR** satisfied the restrictions made by secondary arbiters. After all, *post-condition checkers* check equivalence between corresponding **MR** and **CR**. If the **MR** and **CR** are equal, the test process goes on. If there is a problem with messages, the test system shows a given error and stops working.

Test successfully finishes when the stimuli generators makes everything it was asked to do by the test developer (like visiting all reachable states of the FSM, etc.).

#### IV. ARCHITECTURE OF MERGED TEST SYSTEM

The proposed TLM-based approach to develop single test system can be used while the task is to make test system for a total DUV on the base of test systems for the components of the DUV. To reuse test systems for components is convenient when parts of test systems can be connected to each other by means of the interfaces they have. Therefore, the selected TLM-based way of interface development has a certain advantage: it allows reusing components of test systems as they are, not taking only parts from components or using copy-paste method. To develop complex test system is possible by means of the following steps.

When there are several test systems to connect some of them will miss their connection with DUV. We propose to create common test system and inject all small reaction checkers from earlier developed test systems connected to each other (see Figure 4). Therefore, input and output interfaces adapters and reaction detectors should be modified to connect with other reaction checkers. Fortunately, their separation from the reference model allows us to do without huge reference model modifications.

When merging the reaction checkers, we create the common test system and place sub reaction checkers into it. The common test system possesses its own stimuli generator, reaction checker and coverage tracker. While developing all these parts, to reuse some parts of test systems previously developed would be great achievement.

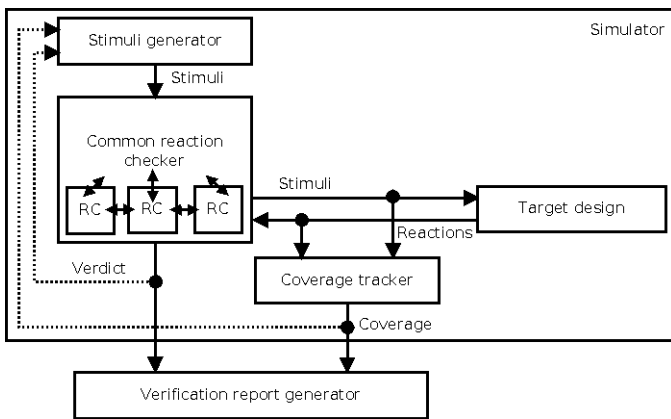


Figure 4. Architecture of multi-module test system

The stimuli generator can inherit scenario functions from sub generators but only if the part of message prepare and call of original reaction checkers were split from each other to different functions. In this case, the reaction checker used can be easily changed to a common by means of overloading the appropriate functions.

The coverage tracker is a common component for all test systems. To use it, a coverage structure should be described and registered in a tracker. The registration is identical in cases of single or multi-module test systems. To refresh the coverage information, some functions from functional models are usually used. Since the models have been inserted into the common test system, the reuse of the coverage becomes free of charge. The common test system just calls all of them at every cycle to make them collect data. It should be noticed, that coverage structures from single test systems in some cases do not provide important information for the case of combined DUV and in this case either cross-coverage is created or new coverage structures for the whole DUV are developed.

The combined reaction checkers is one of the most difficult parts of combined test systems. The common reaction checker looks like the single reaction checker but it should use included reaction checkers functionality. Only the common reaction checker is allowed to change values of DUV's wires while sub reaction checkers' input and output interfaces adapters are switched off. To switch them off is possible by means of overloading to make them send model message not in the absent DUV but to the other reaction checkers (see Figure 5).

To facilitate the connection between reaction checkers we propose to use *channels*. Channel is a way to connect an output interface model and an input interface model together. To do it the message from the input interface should be translated into a form applicable to the output interface to put into it. The channel can also broadcast message into several input interfaces. Usage of channels is given in Figure 6.

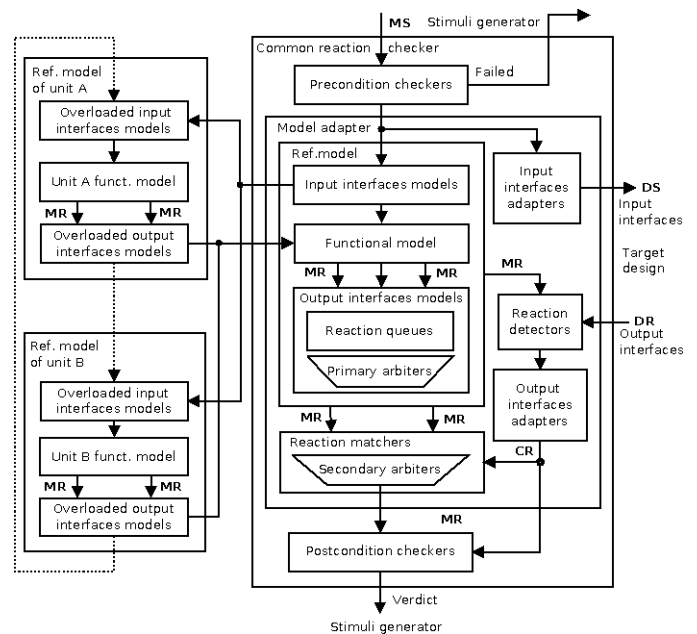


Figure 5. Architecture of multi-module reaction checkers

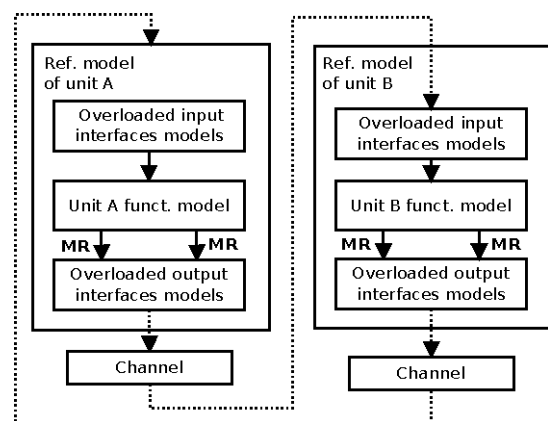


Figure 6. Architecture of multi-module reaction checkers

The overloaded input interfaces models usually contain precondition checkers so that they check protocols of the communications between sub modules. It can help to reveal problems, which can be found if the reference model takes input stimuli only from stimuli generator: the input variables values have wide variance and this variance usually corresponds to the real work situation for the DUV's parts.

The most distinctive feature of the approach, as it has been said before, is that it uses C++ and can use some parts of system simulators usually written in C++. Moreover, the reference models due to their architecture can be reused in development of Verilog-models. In this case, input and output interface adapters do not work as usual: input interface adapters should take the messages from the DUV, process them in functional model and send the messages to the DUV by means output interface adapters (see Figure 7).

Non-blocking L2 cache	Up to detailed-timed	3	6
Northbridge data switch	Up to cycle-accurate	3	3
Memory access unit (MAU)	Up to cycle-accurate	1	1

The labor costs in Table 1 include verification plan writing, test system development, as well as verification process and debug of the developed test system. In case of non-blocking L2 cache the costs also include test system maintaining due to permanent modifications in the DUT. Table 1 shows that time spent to the verification can be roughly estimated to be one man-month per one KLOC. The L2 cache case is an exception from this rule, but it required additional supporting as it has been already said.

A comparison between our results and the second approach (OVM) could be worth knowing. Very little estimations of the OVM application efforts prevented us from doing it. We suggested those spent to the development of test systems with close functionality to be similar to the proposed methodology. It is because OVM also utilizes object-oriented language and the set of test system components looks like proposed. Nevertheless, there is a certain difference between approach given in this article and OVM: our approach provides additional means of FSM-based stimuli flow creation. Actually, this question has not been thoroughly analyzed by us. It is a point of the following research.

The proposed way of merging is a new revealed ability of the basic approach. Only some experiments were conducted to estimate the possibility of merging. First, the test system for a simple FIFO module was developed. It took about two men-days including efforts spent for documenting of the project. Then this FIFO module was multiplied to become three FIFO-modules into one envelop. Two of them were input buffers and the third one was output buffer. We placed between them an arbiter to select which one of the input buffers sends data to the output one. It always selected the first FIFO if it contained any data. To test this combination we combined three test systems for original FIFO-module and added functionality of the arbiter in a very simple way: functional model always reads data from the first FIFO if it contained any data and in other case read the second FIFO. Stimuli generator used original scenario functions like “pop” and “push” but with a little modification as now two FIFO played role of the input and the last one did of the output. The interface adapters of sub test systems were overloaded. Formerly, to make pop and push operations only two interfaces had been used: one input and out output. In case of input FIFOs, input interfaces were not changed (the functionality of setting values to the DUV’s wires from sub reaction checkers had been removed before). Output interfaces were overloaded to send messages into output FIFO’s input interfaces. Initially, the registration of adapters of interfaces had looked like:

```
CPPTESK_SET_OUTPUT_ADAPTER(iface3,
    FIFOMediator::deserialize_iface3);
```

where deserializer is a function which translates DUV’s wires signals into message. As the output interface iface3 was not already output one, we overloaded this adapter:

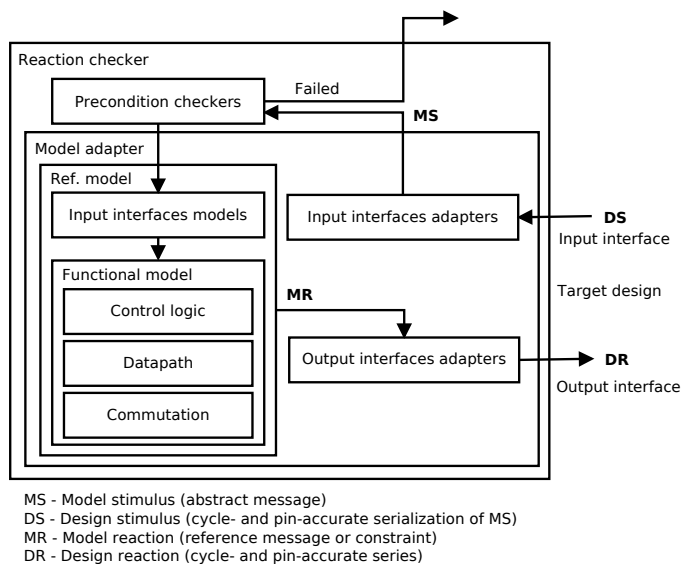


Figure 7. Architecture of reaction checker built into DUV

Common test system controls the reaction checkers built in DUV. The checkers help Verilog-model developer in accelerated development of the model as code in C++ with the same functionality as in Verilog is usually written quicker.

Summing up, the approach allows developing test systems, which can be reused as parts of common test system. These test systems check not only the output data of DUV but input data from, say, stimuli generator (by means of precondition checkers). When the test systems are connected to each other not to DUV, they can check the behavior of their neighbors, i.e. they check the interconnection protocol of DUV’s components. The test system supports special means to make the interconnection such as interfaces and channels. After all, to reduce the time spent to make the first version of DUV for system-level verification, the test systems can be inserted into the Verilog code of DUV while the last is still under designing. The approach is supported by a library developed in C++, so that test systems being developed according to the approach, are also based on C++. It gives wide range of opportunities to use all means of C++ to facilitate the development of the test systems. It should be noticed, that C++ is usually used in system level simulators of DUV, so that parts of the simulators can be easily reused as reference models in the test systems and vice versa. At last, the library is compatible with UniTESK approach, which means the opportunity to develop high-quality tests based on FSM-traversing even for the system-level case and to spread test systems among clusters of computers.

## V. CASE STUDY

The approach to develop single test systems has shown its effectiveness in several projects [4]. The most interesting cases are generalized in Table 1.

TABLE I. APPLICATIONS OF THE SUGGESTED APPROACH

Design under verification	Depth of verification	Source code, KLOC	Labor costs, man-months
Translation lookaside buffer (TLB)	Up to cycle-accurate	2.5	2.5

```

CPPTESK_SET_INNER_IFACE_ADAPTER(FIFOMediator,
fifo0, fifo0.iface3, MM::deserialize_inner_iface0);

```

New deserializer calls the output FIFO fifo2:

```

CPPTESK_DEFINE_PROCESS(
    MM::deserialize_inner_iface0)
{
    ...
    if(!fifo2.is_full()) {
        ...
        fifo2.start(&FIFO::push_msg,
                    fifo2.iface1, data);
    }
}

```

The output interface model of output FIFO just gave the messages into common reaction checker's output interface model. To create the common test system we spent about half a day accounting the time to research of merging possibility. The time could have been spent for developing of complex test system from scratch is expected to be about 2 days, but it is not the most important. The time to connect sub test systems slightly correlates with complexity of DUV's sub part. Mostly, it depends on amount of input and output interfaces and efforts to connect them together. We have an estimate that to connect exactly one input interface to one output is possible in about one hour. This time will be spent to develop channel between them that will translate messages between channels. The average DUV's part, by our estimates, consists of about ten input and ten output interfaces, so that to connect two average DUV's parts is possible in one-two men-days, according to the productivity of the verification engineer.

## VI. CONCLUSIONS

This approach gives us a useful way to develop test systems for separated parts of DUVs and then to merge the test systems with high level of reuse. The main advantages of the approach are the check of interconnection behavior with no additional code and special means to facilitate reuse. The architecture is

supported by a library developed in C++ that allows using the great opportunities of the language while developing test systems. The fact that usually system level simulators are based on C++ points to the ability of reusing parts of system-level reference models and reference models of the developed test systems and vice versa. There are two bonuses: the test systems can control communications between functional models inside the DUV as its parts, especially when the DUV is still under construction, and that the library supporting the approach supports UniTESK technology which allows developing high-quality tests based on FSM-traversing and spreading test systems among clusters of computers.

## REFERENCES

- [1] IEEE 1364-2005, Verilog Standard.
- [2] J. Bergeron, Writing testbenches: functional verification of HDL models. Kluwer Academic Publishers, 2003.
- [3] Tool called Formality by Synopsys (<http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx>)
- [4] M. Chupilko, A. Kamkin. A TLM-based approach to functional verification of hardware components at different abstraction levels. 12<sup>th</sup> Latin-American Test Workshop, 2011.
- [5] E. Clarke, O. Grumberg, D. Peled. Model Checking. MIT Press, 1999.
- [6] R. Kneuper. Limits of Formal Methods. Formal Aspects of Computing (1997) 3: 1-000, 1997.
- [7] S. Iman. Step-by-step Functional Verification with SystemVerilog and OVM. Hansen Brown Publishing Company, 2008.
- [8] V. Ivannikov, A. Kamkin, V. Kuliain, A. Petrenko. Application of the UniTESK technology to functional verification of software. [http://citforum.ru/SE/testing/unitesk\\_hard/](http://citforum.ru/SE/testing/unitesk_hard/), 2006 (in Russian)
- [9] Open Verification Methodology, <http://www.ovmworld.org>.
- [10] Y. Gubenko, A. Kamkin, M. Chupilko. Comparative analysis of modern technologies of hardware designs test development. [http://citforum.ru/SE/testing/hardware\\_models/](http://citforum.ru/SE/testing/hardware_models/), 2009 (in Russian)
- [11] A. Barancev et al. UniTesK approach to the software development: achievements and prospects. <http://citforum.ru/SE/testing/unitesk/>, 2004. (in Russian)

# Programming for Modular Reconfigurable Robots

Anna Gorbenko  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: gorbenko.aa@gmail.com

Vladimir Popov  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: Vladimir.Popov@usu.ru

**Abstract**—Composed of multiple modular robotic units, self-reconfigurable modular robots are metamorphic systems that can autonomously rearrange the modules and form different configurations for dynamic environments and tasks. Self-reconfiguration is to solve how to change connectivity among modules to transform the robot from the current configuration into the goal configuration within the restrictions of physical implementation. The existing reconfiguration algorithms used different methods, such as divide-and-conquer, graph matching etc, to reduce the reconfiguration cost. However, the optimal solution with least reconfiguration steps has never been reached. The optimal reconfiguration planning problem of finding the least number of reconfiguration steps to transform between two configurations is NP-complete. In this paper we describe an approach to solve this problem. This approach is based on constructing a logical models for considered problem.

## I. INTRODUCTION

Modular robotics has been the subject of much interest in the research community [1]. Using large numbers of simple modules to replace one complicated, special-purpose device provides benefits in terms of flexibility, robustness, and manufacturing cost. The challenge in these systems lies in controlling large numbers of low-powered, unreliable modules. Motion planning and shape formation for these systems is the main problem of such a difficult challenge.

Metamorphic robotic systems [2] can be viewed as a large swarm of connected robots which collectively act as a single entity. Potential applications of metamorphic systems composed of a large number of modules include:

- obstacle avoidance in highly constrained and unstructured environments;
- growing structures composed of modules to form bridges, buttresses, and other civil structures in times of emergency;
- envelopment of objects, such as recovering satellites from space;
- performing inspections in constrained environments such as nuclear reactors.

Self-reconfiguring robots were first proposed in [3]. In this planar system modules were heterogeneous and semi-autonomous. Other research focused on homogeneous systems with non-autonomous modules in two dimensions [4] – [7] and three dimensions [8] – [10]. In this type of system the modules are not capable of acting independently, and thus

must remain connected. Five types of modular reconfigurable robotic systems have been proposed in the literature:

- robots in which modules are reconfigured using external intervention, e.g. [11] – [14];
- cellular robotic systems in which a heterogeneous collection of independent specialized modules are coordinated, e.g. [15] – [18];
- swarm intelligence in which there are generally no physical connections between modules, e.g. [19] – [22];
- modular robots composed of a few basic elements which can be composed into complex systems and used for various modes of locomotion, e.g. [23] – [25];
- fractal systems composed of modules with zero kinematic mobility, but which can walk over each other in discrete quanta due to changes in the polarity of magnetic fields, e.g. [5], [26].

In the present work, a metamorphic robotic system is a collection of independently controlled mechatronic modules, each of which has the ability to connect, disconnect, and climb over adjacent modules, e.g. [6]. A metamorphic system can dynamically reconfigure by the locomotion of modules over their neighbors. Thus they can be viewed as a collection of connected modular robots which act together to perform the given task. Composed of multiple modular robotic units, self-reconfigurable modular robots are metamorphic systems that can autonomously rearrange the modules and form different configurations for dynamic environments and tasks.

Modular reconfigurable robot programming can be substantially more challenging than normal robot programming due to:

- scale / number of modules;
- concurrency and asynchronicity, both in physical interactions and potentially at the software level;
- the local scope of information naturally available at each module.

For modular reconfigurable robots it is developed several specialized programming languages (e.g. [27], [28]). However, existing programming methods show relatively poor performance for reconfiguration planning problems. Note that reconfiguration planning problems play a central role for modular robots (e.g. [29] – [40]). Solutions for such problems lies at the heart of any control system of modular robots. Performance

of such solutions is the base factor for the performance of the whole control system. So, the main challenges for modular robotic systems is an efficient planner. It has long been recognized that traditional methods are unsuitable due to the large search space and the blocking constraints imposed by realizable module design. To ease the planning problem, many groups have proposed different kinds of metamodules, groups of modules that act as a unit for planning or motion execution purposes, each specific to a particular module design [7], [38], [41], [42].

Poor performance for reconfiguration planning problems is not surprising, since such problems are computationally hard. In particular, it is proved that the optimal reconfiguration planning problem of finding the least number of reconfiguration steps to transform between two configurations (ORP) is **NP**-complete. Therefore, we need some intelligent solution for this problem. However, the applying of distributed algorithms or any iterative procedure requires a great exchange of information between modules. This leads to the loss of solution accuracy and reduce performance. Therefore, it is desirable to solve ORP in a separate intelligent module which would generate a final solution represented by simple instructions. Note that the centralization of ORP solution allows to use some remote computing resources and makes the performance independent from computing resources of modules. When using such approach, programming of individual modules consists in

- processing of sensory information;
- transmission of sensory information;
- motor control;
- receiving instructions for actuators.

In this paper we describe an approach to solve ORP problem. This approach is based on constructing a logical models for considered problem.

## II. OPTIMAL RECONFIGURATION PLANNING PROBLEM

Self-reconfiguration is to solve how to change connectivity among modules to transform the robot from the current configuration into the goal configuration within the restrictions of physical implementation. Depending on the hardware design, reconfiguration algorithms fall into two groups:

- reconfiguration for lattice-type modular robot and reconfiguration for chain-type modular robot. In lattice-type robot, modules lie in 2D or 3D grids;
- the reconfiguration is achieved through discrete movements of modules detaching from the current lattice location, moving along and surface of the robot and docking at the adjacent cells.

Example reconfiguration work includes [32] – [40] etc. In chain-type robots, modules can form moving chains and loops of any graph topology, and the reconfiguration is achieved through “connect” and “disconnect” operations between modules along with the joint motion of chains composed of several modules. Due to its difficulty, the chain-type reconfiguration has received less attention. Existing algorithms include [43] –

[47] etc. The different geometric arrangement of modules between lattice-type and chain-type modular robots makes their reconfiguration planning mechanisms fundamentally different. The work in this paper is more focused on chain-type reconfiguration. For simplicity, we will use the term “modular robots” or simply “robots” to denote “the chain-type modular robots”, and use “reconfiguration” to denote “chain-type reconfiguration” in the following.

The existing reconfiguration algorithms used different methods, such as divide-and-conquer [43], graph matching [44] etc, to reduce the reconfiguration cost. However, the optimal solution with least reconfiguration steps has never been reached. In [48] proved that the optimal reconfiguration planning problem of finding the least number of reconfiguration steps to transform between two configurations is **NP**-complete.

### A. Configuration Representation

Before defining the optimal reconfiguration planning problem, we would describe representation of robot’s configuration first. Two robots with the same graph topology can function differently if the modules are connected via different connectors (see e.g. [48]). To fully represent a robot’s configuration, a special graph called C-Graph (Connector-Graph) is proposed in [48]. C-Graph is the extension of normal graph with differentiated connecting points. Each node has a finite number of ports that are internally labeled corresponding to the connectors of a module. A connection between module  $u$ ’s connector  $i$  and module  $v$ ’s connector  $j$  corresponds to an edge  $(i, j)$  between  $u$  and  $v$ .

In principle we could represent a robot’s configuration as a C-Graph

$$G = (V, E),$$

$$V = \{v[1], v[2], \dots, v[n]\},$$

$$E = \{e[1], e[2], \dots, e[m]\},$$

where:

- each node  $v[i] \in V$  represents the set

$$v[i] = \{v[i, 1], v[i, 2], \dots, v[i, p_i]\}$$

of connecting points of  $i$ th module, where  $p_i$  is the number of connecting points of  $i$ th module;

- each edge

$$e[j] = (v[i_1, l_1], v[i_2, l_2]) \in E$$

represents a connection between module  $i_1$ ’s connector  $l_1$  and module  $i_2$ ’s connector  $l_2$ , where

$$1 \leq i_1 \leq n, 1 \leq i_2 \leq n,$$

$$1 \leq l_1 \leq p_{i_1}, 1 \leq l_2 \leq p_{i_2}.$$

## B. Reconfiguration Actions

The two elementary reconfiguration actions are:

- making new connections;
- disconnecting current connections between modules for connectivity rearrangement.

The robot can bend its body through module joints, so any two modules with free connectors can potentially be aligned and dock with each other.

## C. Optimal Reconfiguration Planning Problem

The reconfiguration planning problem is defined as how modules in one configuration rearrange into another using several sets of reconfiguration actions. Basically, what connections to make and what connections to disconnect so as to reconfigure from arbitrary one shape to another? Without loss of generality, we will always assume that the number of modules in the initial configuration is the same as that in the goal configuration.

During the reconfiguration process, the reconfiguration actions are most time- and energy-consuming, so it is a common practice to aim at minimizing the number of reconfiguration steps, i.e. the number of connect actions plus the number of disconnect actions. Therefore, the optimal reconfiguration planning problem is to find the least number of reconfiguration steps to transform from the initial configuration into the goal configuration.

Since the number of physical connections is predefined in the initial and goal configurations, the number of connect actions is fixed once the number of disconnect action is known, and vice versa. So we get that the optimal reconfiguration planning problem is to find the either one of the following metrics:

- least number of connect actions;
- least number of disconnect actions;
- least number of reconfiguration steps (i.e., the number of connect actions plus the number of disconnect actions).

For given two connected C-Graphs

$$I = (V, E_1)$$

and

$$G = (V, E_2)$$

we say that there exists a reconfiguration plan with at most  $k$  reconfiguration steps if and only if there exists a sequence of  $r \leq k$  connect and disconnect actions such that starting from  $I$  we obtain  $G$  and applying each of this connect and disconnect actions we obtain a connected C-Graph. The decision version of optimal reconfiguration planning problem is formulated as the following problem.

OPTIMAL RECONFIGURATION PLANNING PROBLEM (ORP):

INSTANCE: C-Graphs  $I = (V, E_1)$  and  $G = (V, E_2)$ , a given integer  $k$ .

QUESTION: Whether there exists a reconfiguration plan for C-Graphs  $I$  and  $G$  with at most  $k$  reconfiguration steps?

## III. LOGICAL MODEL OF ORP

The propositional satisfiability problem (PSAT) is a core problem in mathematical logic and computing theory. Propositional satisfiability is the problem of determining if the variables of a given boolean function can be assigned in such a way as to make the formula evaluate to true. PSAT was the first known **NP**-complete problem, as proved by Stephen Cook in 1971 [49]. Until that time, the concept of an **NP**-complete problem did not even exist. Considered also different variants of the satisfiability problem. For instance, Satisfiability (SAT) is the problem of determining if the variables of a given boolean function in conjunctive normal form can be assigned in such a way as to make the formula evaluate to true. In practice, the satisfiability problem is fundamental in solving many problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design, computer architecture design, and computer network design. Traditional methods treat the satisfiability problem as a discrete, constrained decision problem.

### A. Reduction to PSAT

Consider a set of C-Graphs

$$\{G[q] = (V, E[q] \mid 0 \leq q \leq k)\},$$

where

$$E[q] = \{e[q, 1], e[q, 2], \dots, e[q, m_q]\},$$

each edge

$$e[q, j] = (v[i_1, l_1], v[i_2, l_2]) \in E[q]$$

represents a connection between module  $i_1$ 's connector  $l_1$  and module  $i_2$ 's connector  $l_2$ , where

$$1 \leq i_1 \leq n, 1 \leq i_2 \leq n, 1 \leq l_1 \leq p_{i_1}, 1 \leq l_2 \leq p_{i_2}.$$

Let  $G[0] = I$ ,  $G[k] = G$ . Now consider a set of boolean variables

$$\{x[q, i_1, i_2, i_3, i_4] \mid 0 \leq q \leq k, 1 \leq i_1 \leq n,$$

$$1 \leq i_2 \leq p_{i_1}, 1 \leq i_3 \leq n, 1 \leq i_4 \leq p_{i_3}\}.$$

Suppose that relation

$$x[q, i_1, i_2, i_3, i_4] = 1$$

means that

$$(v[i_1, i_2], v[i_3, i_4]) \in E[q].$$

Consider following boolean function:

$$\psi[q] \Leftrightarrow \bigvee_{\substack{1 \leq s_1 \leq n, \\ 1 \leq s_2 \leq p_{i_1}, \\ 1 \leq s_3 \leq n, \\ 1 \leq s_4 \leq p_{i_3}}} \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ i_1 \neq s_1, \\ i_2 \neq s_2, \\ i_3 \neq s_3, \\ i_4 \neq s_4, \\ i_1 \neq s_3, \\ i_2 \neq s_4, \\ i_3 \neq s_1, \\ i_4 \neq s_2}} x[q, i_1, i_2, i_3, i_4] =$$

$$x[q+1, i_1, i_2, i_3, i_4].$$

It is easy to see that boolean function  $\psi[q]$  is satisfiable if and only if  $G[q] = G[q+1]$  or C-Graph  $G[q+1]$  obtained from  $G[q]$  by one connect or disconnect action. Therefore, it is easy to see that boolean function

$$\left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \in E[0]} x[0, i_1, i_2, i_3, i_4] = 1 \right) \wedge$$

$$\left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \notin E[0]} x[0, i_1, i_2, i_3, i_4] = 0 \right) \wedge$$

$$\left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \in E[k]} z[k, i_1, i_2, i_3, i_4] = 1 \right) \wedge$$

$$\left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \notin E[k]} z[k, i_1, i_2, i_3, i_4] = 0 \right) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \left( \bigvee_{j=1}^{n^2} w[j, i_1, i_2, i_3, i_4] \right) \right) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \left( \bigwedge_{\substack{1 \leq j_1 \leq n^2, \\ 1 \leq j_2 \leq n^2, \\ j_1 \neq j_2}} (\neg w[j_1, i_1, i_2, i_3, i_4] \vee$$

$$\neg w[j_2, i_1, i_2, i_3, i_4])) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq p_{i_1}, \\ 1 \leq i_7 \leq n, \\ 1 \leq i_8 \leq p_{i_3}, \\ (i_1, i_3) \neq (i_5, i_7), \\ (i_1, i_3) \neq (i_7, i_5)}} \bigvee_{j=1}^{n^2} (\neg w[j, i_1, i_2, i_3, i_4] \vee$$

$$\neg w[j, i_5, i_6, i_7, i_8])) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \bigvee_{j=1}^{n^2} w[j, i_1, i_2, i_3, i_4] =$$

$$w[j, i_3, i_2, i_1, i_4]) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \rightarrow$$

$$x[k, i_1, i_2, i_3, i_4] = z[k, i_5, i_2, i_6, i_4]) \wedge$$

$$\left( \bigwedge_{0 \leq q \leq k-1} \psi[q] \right)$$

is satisfiable if and only if there exists a reconfiguration plan for C-Graphs  $I$  and  $G$  with at most  $k$  reconfiguration steps.

Note that

$$(\alpha = \beta) \Leftrightarrow ((\alpha \vee \neg \beta) \wedge (\neg \alpha \vee \beta)).$$

Therefore,  $\psi[q] \Leftrightarrow \psi'[q]$ , where

$$\psi'[q] \Leftrightarrow \bigvee_{\substack{1 \leq s_1 \leq n, \\ 1 \leq s_2 \leq p_{i_1}, \\ 1 \leq s_3 \leq n, \\ 1 \leq s_4 \leq p_{i_3}}} \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ i_1 \neq s_1, \\ i_2 \neq s_2, \\ i_3 \neq s_3, \\ i_4 \neq s_4, \\ i_1 \neq s_3, \\ i_2 \neq s_4, \\ i_3 \neq s_1, \\ i_4 \neq s_2}} ((x[q, i_1, i_2, i_3, i_4] \vee$$

$$\neg x[q+1, i_1, i_2, i_3, i_4]) \wedge$$



$$\begin{aligned}
& (\neg x[q, i_1, i_2, i_3, i_4] \vee \\
& x[q + 1, i_1, i_2, i_3, i_4]), \\
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \bigwedge_{j=1}^{n^2} w[j, i_1, i_2, i_3, i_4] = \\
& w[j, i_3, i_2, i_1, i_4] \Leftrightarrow \\
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \bigwedge_{j=1}^{n^2} ((w[j, i_1, i_2, i_3, i_4] \vee \\
& \neg w[j, i_3, i_2, i_1, i_4]) \wedge \\
& (\neg w[j, i_1, i_2, i_3, i_4] \vee \\
& w[j, i_3, i_2, i_1, i_4])).
\end{aligned}$$

Since  $\alpha \rightarrow \beta \Leftrightarrow \neg\alpha \vee \beta$ ,

$$\begin{aligned}
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \rightarrow \\
& x[k, i_1, i_2, i_3, i_4] = z[k, i_5, i_2, i_6, i_4] \Leftrightarrow \\
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} \neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \\
& x[k, i_1, i_2, i_3, i_4] = z[k, i_5, i_2, i_6, i_4] \Leftrightarrow \\
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} (\neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \\
& ((x[k, i_1, i_2, i_3, i_4] \vee \\
& \neg z[k, i_5, i_2, i_6, i_4]) \wedge \\
& (\neg x[k, i_1, i_2, i_3, i_4] \vee \\
& z[k, i_5, i_2, i_6, i_4])).
\end{aligned}$$

So, using only  $\neg$ ,  $\wedge$ , and  $\vee$ , we obtain a boolean function

$$\begin{aligned}
\xi_1 \Leftrightarrow & \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \in E[0]} x[0, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \notin E[0]} \neg x[0, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \in E[k]} z[k, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \notin E[k]} \neg z[k, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \left( \bigvee_{j=1}^{n^2} w[j, i_1, i_2, i_3, i_4] \right) \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \left( \bigwedge_{\substack{1 \leq j_1 \leq n^2, \\ 1 \leq j_2 \leq n^2, \\ j_1 \neq j_2}} (\neg w[j_1, i_1, i_2, i_3, i_4] \vee \right. \right. \\
& \left. \left. \neg w[j_2, i_1, i_2, i_3, i_4]) \right) \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq p_{i_1}, \\ 1 \leq i_7 \leq n, \\ 1 \leq i_8 \leq p_{i_3}, \\ (i_1, i_3) \neq (i_5, i_7), \\ (i_1, i_3) \neq (i_7, i_5)}} \bigwedge_{j=1}^{n^2} (\neg w[j, i_1, i_2, i_3, i_4] \vee \right. \\
& \left. \neg w[j, i_5, i_6, i_7, i_8]) \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \bigwedge_{j=1}^{n^2} ((w[j, i_1, i_2, i_3, i_4] \vee \\
& \neg w[j, i_3, i_2, i_1, i_4]) \wedge \\
& (\neg w[j, i_1, i_2, i_3, i_4] \vee \\
& w[j, i_3, i_2, i_1, i_4])) \right) \wedge
\end{aligned}$$



$$\begin{aligned}
& (\neg x[q, t_1, t_2, t_3, t_4] \vee \\
& x[q + 1, t_1, t_2, t_3, t_4]) \Leftrightarrow \\
& (x[q, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& x[q, t_1, t_2, t_3, t_4] \vee \\
& \neg x[q + 1, t_1, t_2, t_3, t_4]) \wedge \\
& (\neg x[q, s_1, s_2, s_3, s_4] \vee \\
& x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& x[q, t_1, t_2, t_3, t_4] \vee \\
& \neg x[q + 1, t_1, t_2, t_3, t_4]) \wedge \\
& (x[q, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q, t_1, t_2, t_3, t_4] \vee \\
& x[q + 1, t_1, t_2, t_3, t_4]) \wedge \\
& (\neg x[q, s_1, s_2, s_3, s_4] \vee \\
& x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q, t_1, t_2, t_3, t_4] \vee \\
& x[q + 1, t_1, t_2, t_3, t_4]).
\end{aligned}$$

Therefore,  $\psi'''[q] \Leftrightarrow \psi''''[q]$ , where

$$\begin{aligned}
\psi''''[q] \Leftrightarrow & \bigwedge_{\substack{1 \leq s_1 \leq n, \\ 1 \leq s_2 \leq p_{i_1}, \\ 1 \leq s_3 \leq n, \\ 1 \leq s_4 \leq p_{i_3}, \\ 1 \leq t_1 \leq n, \\ 1 \leq t_2 \leq p_{i_1}, \\ 1 \leq t_3 \leq n, \\ 1 \leq t_4 \leq p_{i_3}, \\ (s_1, s_2, s_3, s_4) \neq (t_1, t_2, t_3, t_4), \\ (s_1, s_2, s_3, s_4) \neq (t_3, t_4, t_1, t_2),}} (x[q, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q + 1, s_1, s_2, s_3, s_4] \vee x[q, t_1, t_2, t_3, t_4] \vee \\
& \neg x[q + 1, t_1, t_2, t_3, t_4]) \wedge \\
& (\neg x[q, s_1, s_2, s_3, s_4] \vee \\
& x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& x[q, t_1, t_2, t_3, t_4] \vee \\
& \neg x[q + 1, t_1, t_2, t_3, t_4]) \wedge \\
& (x[q, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q, t_1, t_2, t_3, t_4] \vee \\
& x[q + 1, t_1, t_2, t_3, t_4]) \wedge
\end{aligned}$$

$$\begin{aligned}
& (\neg x[q, s_1, s_2, s_3, s_4] \vee \\
& x[q + 1, s_1, s_2, s_3, s_4] \vee \\
& \neg x[q, t_1, t_2, t_3, t_4] \vee \\
& x[q + 1, t_1, t_2, t_3, t_4]).
\end{aligned}$$

Note that

$$\begin{aligned}
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} (\neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \\
& ((x[k, i_1, i_2, i_3, i_4] \vee \\
& \neg z[k, i_5, i_2, i_6, i_4]) \wedge \\
& (\neg x[k, i_1, i_2, i_3, i_4] \vee \\
& z[k, i_5, i_2, i_6, i_4]))) \Leftrightarrow \\
& \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} ((\neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \\
& x[k, i_1, i_2, i_3, i_4] \vee \\
& \neg z[k, i_5, i_2, i_6, i_4]) \wedge \\
& (\neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \\
& \neg x[k, i_1, i_2, i_3, i_4] \vee \\
& z[k, i_5, i_2, i_6, i_4])).
\end{aligned}$$

So, we obtain a boolean function

$$\begin{aligned}
\xi_2 \Leftrightarrow & \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \in E[0]} x[0, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \notin E[0]} \neg x[0, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \in E[k]} z[k, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{(v[i_1, i_2], v[i_3, i_4]) \notin E[k]} \neg z[k, i_1, i_2, i_3, i_4] \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3},}} \left( \bigvee_{j=1}^{n^2} w[j, i_1, i_2, i_3, i_4] \right) \right) \wedge
\end{aligned}$$

$$\begin{aligned}
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3},}} \left( \bigwedge_{\substack{1 \leq j_1 \leq n^{n^2}, \\ 1 \leq j_2 \leq n^{n^2}, \\ j_1 \neq j_2}} (\neg w[j_1, i_1, i_2, i_3, i_4] \vee \right. \right. \\
& \quad \left. \left. \neg w[j_2, i_1, i_2, i_3, i_4])) \right) \wedge \right. \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq p_{i_1}, \\ 1 \leq i_7 \leq n, \\ 1 \leq i_8 \leq p_{i_3}, \\ (i_1, i_3) \neq (i_5, i_7), \\ (i_1, i_3) \neq (i_7, i_5)}} \bigwedge_{j=1}^{n^2} (\neg w[j, i_1, i_2, i_3, i_4] \vee \right. \\
& \quad \left. \neg w[j, i_5, i_6, i_7, i_8])) \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}}} \bigwedge_{j=1}^{n^2} ((w[j, i_1, i_2, i_3, i_4] \vee \right. \\
& \quad \neg w[j, i_3, i_2, i_1, i_4]) \wedge \\
& \quad (\neg w[j, i_1, i_2, i_3, i_4] \vee \\
& \quad w[j, i_3, i_2, i_1, i_4])) \right) \wedge \\
& \left( \bigwedge_{\substack{1 \leq i_1 \leq n, \\ 1 \leq i_2 \leq p_{i_1}, \\ 1 \leq i_3 \leq n, \\ 1 \leq i_4 \leq p_{i_3}, \\ 1 \leq i_5 \leq n, \\ 1 \leq i_6 \leq n}} ((\neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \right. \\
& \quad x[k, i_1, i_2, i_3, i_4] \vee \\
& \quad \neg z[k, i_5, i_2, i_6, i_4]) \wedge \\
& \quad (\neg w[(i_5 - 1)n + i_6, i_1, i_2, i_3, i_4] \vee \\
& \quad \neg x[k, i_1, i_2, i_3, i_4] \vee \\
& \quad z[k, i_5, i_2, i_6, i_4])) \right) \wedge \\
& \left( \bigwedge_{0 \leq q \leq k-1} \psi'''[q] \right)
\end{aligned}$$

such that  $\xi_2$  is satisfiable if and only if there exists a reconfiguration plan for C-Graphs  $I$  and  $G$  with at most  $k$  reconfiguration steps. It is easy to see that the size of boolean function  $\xi_2$  polynomially depends from the size of C-Graphs. Since  $\xi_2$  in conjunctive normal form, we obtain an explicit reduction from ORP to SAT.

#### IV. CONCLUSION AND EXPERIMENTAL RESULTS

In recent years, many optimization methods, parallel algorithms, and practical techniques have been developed for solving the satisfiability problem (see [50]). In particular, proposed several genetic algorithms [51] – [54]. Considered hybrid algorithms in which the approach of genetic algorithms combined with local search [55].

Modern propositional satisfiability solvers are usually designed to solve SAT formula encoded in conjunctive normal form (CNF). Stochastic local search techniques have been successful in solving propositional satisfiability problems encoded in CNF. Recently complete solvers have shown that there are advantages to tackling propositional satisfiability problems in a more expressive natural representation, since the conversion to CNF can lose problem structure and introduce significantly more variables to encode the problem. CNF solvers can be disadvantageous for problems which are more naturally encoded as arbitrary propositional formula. The conversion to CNF form may increase the size of the formula exponentially, or significantly reduce the strength of the formulation. The translation may introduce many new variables which increases the size of the raw valuation space through which the solver must search. Recently, interest has arisen in designing non-clausal satisfiability algorithms (see e.g. [56] – [63]).

Relatively high efficiency demonstrated by algorithms based solely on local search. Of course, these algorithms require exponential time at worst. But they can relatively quick receive solutions for many boolean functions. Therefore, it is natural to use a reduction to different variants of the satisfiability problem to solve computational hard problems.

Encoding problems as Boolean satisfiability and solving them with very efficient satisfiability algorithms has recently caused considerable interest. In particular, local search algorithms have given impressive results on many problems. For example, there are several ways of SAT-encoding constraint satisfaction [64] – [73], clique [74], planning [75] – [95], and colouring problems [74], [96] – [98]. The maximum cut, vertex cover and maximum independent set problems can be reduced to MAX-2-SAT [99] – [101]. There are a number of implicit reductions from the Hamiltonian cycle problem to the satisfiability (SAT) problem (see [74], [102], [103]).

In previous section we obtain an implicit reduction from the optimal reconfiguration planning problem of finding the least number of reconfiguration steps to transform between two configurations to some variants of satisfiability: PSAT, SAT. We create a generator of special hard and natural instances for the optimal reconfiguration planning problem of finding the least number of reconfiguration steps to transform between two configurations. We use algorithms from [104]. Also we design our own genetic algorithm for SAT which based on algorithms from [104]. We use heterogeneous cluster based on three clusters (Cluster USU, Linux, 8 calculation nodes, Intel Pentium IV 2.40GHz processors; umt, Linux, 256 calculation nodes, Xeon 3.00GHz processors; um64, Linux, 124 calculation nodes, AMD Opteron 2.6GHz bi-processors)

[105]. For computational experiment we create special hard test sets and natural test sets. Special hard test sets based on ideas from [106]. Natural test sets based on ideas from [48]. In tests we consider systems consisted from approximately 400 of modular robots.

Each test was run on a cluster of at least 100 nodes. For special hard test sets: the maximum solution time was 16 hours; the average time to find a solution was 33.2 minutes; the best time was 116 seconds. For natural test sets: the maximum solution time was 9 hours; the average time to find a solution was 9.8 minutes; the best time was 9 seconds. Based on our experiments we can say that considered model can be used as an efficient planner.

#### ACKNOWLEDGMENT

The work was partially supported by Grant of President of the Russian Federation MD-1687.2008.9 and Analytical Departmental Program "Developing the scientific potential of high school" 2.1.1/1775.

#### REFERENCES

- [1] Rus, D., and Chirikjian, G., Eds., *Special Issue on Self-Reconfiguring Robots*, Autonomous Robotics, 2001, 10(1).
- [2] Chirikjian, G.S. Kinematics of a Metamorphic Robotic System. *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 449–455, San Diego, 1994. IEEE Computer Society Press.
- [3] Fukuda, T., and Nakagawa, S. A Dynamically Reconfigurable Robotic System (Concept of a System and Optimal Configurations). *Proceedings of the 1987 IEEE International Conference on Industrial Electronics, Control, and Instrumentation*, pages 588–595, Los Alamitos, 1987. IEEE Computer Society Press.
- [4] Hosokawa, K., Tsujimori, T., Fujii, T., Kaetsu, H., Asama, H., Kuroda, Y., and Endo, I. Self-Organizing Collective Robots with Morphogenesis in a Vertical Plane. *Proceedings of the 1998 IEEE International Conference on Robotics and Automation. Vol. 4*, pages 2858–2863, Leuven, 1998. IEEE Computer Society Press.
- [5] Murata, S., Kurokawa, H., and Kokaji, S. Self-Assembling Machine. *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 442–448, San Diego, 1994. IEEE Computer Society Press.
- [6] Pamecha, A., Chiang, C., Stein, D., and Chirikjian, G. Design and implementation of metamorphic robots. *Proceedings of The 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference*, pages 1–10, Irvine, 1996. ASME Press.
- [7] Rus, D., and Vona, M. Crystalline Robots: Self-reconfiguration with Unit-compressible Modules. *Autonomous Robots*, 10(1):107–124, 2001.
- [8] Kotay, K., and Rus, D. Locomotion versatility through selfreconfiguration. *Robotics and Autonomous Systems*, 26(2-3):217–232, 1999.
- [9] Murata, S., Yoshida, E., Kamimura, A., Kurokawa, H., Tomita, K., and Kokaji, S. M-TRAN: Self-Reconfigurable Modular Robotic System. *IEEE/ASME Transactions on Mechatronics*, 7(4):431–441, 2002.
- [10] Suh, J., Homans, S., and Yim, M. Telecubes: Mechanical Design of a Module for Self-Reconfiguring Robotics. *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, pages 4095–4101, Washington, 2002. IEEE Computer Society Press.
- [11] Benhabib, B., Zak, G., and Lipton, M.G. A Generalized Kinematic Modeling Method for Modular Robots. *Journal of Robotic Systems*, 6(5):545–571, 1989.
- [12] Cohen, R., Lipton, M.G., Dai, M.Q., and Benhabib, B. Conceptual Design of a Modular Robot. *ASME Journal of Mechanical Design*, 114:117–125, 1992.
- [13] Sciaky, M. Modular Robots Implementation. In Nof, S., editor, *Handbook of Industrial Robotics*, pages 759–774. John Wiley and Sons, 1985.
- [14] Wurst, K.H. The Conception and Construction of a Modular Robot System. In Van Brussel, H., editor, *16th International Symposium On Industrial Robots. 8th International Conference On Industrial Robot Technology: Proceedings*, pages 37–44, Brussels, 1986. Springer-Verlag.
- [15] Beni, G. Concept of Cellular Robotic Systems. *Proceedings of the IEEE International Symposium on Intelligent Control*, pages 57–62, Arlington, 1988. IEEE Computer Society Press.
- [16] Beni, G., and Wang, J. Theoretical Problems for the Realization of Distributed Robotic Systems. *Proceedings of the 1991 IEEE Conference on Robotics and Automation*, pages 1914–1920, Sacramento, 1991. IEEE Computer Society Press.
- [17] Fukuda, T., and Nakagawa, S. Dynamically Reconfigurable Robotic Systems. *Proceedings of the 1988 IEEE Conference on Robotics and Automation*, pages 1581–1586, Philadelphia, 1988. IEEE Computer Society Press.
- [18] Fukuda, T., and Kawauchi, Y. Cellular Robotic System (CEBOT) as One of the Realization of Self-organizing Intelligent Universal Manipulator. *Proceedings of the 1990 IEEE Conference on Robotics and Automation*, pages 662–667, Cincinnati, 1990. IEEE Computer Society Press.
- [19] Beni, G., and Hackwood, S. Stationary Waves in Cyclic Swarms. *Proceedings of IEEE International Symposium on Intelligent Control*, pages 234–242, Los Alamitos, 1992. IEEE Computer Society Press.
- [20] Beni, G., and Wang, J. Swarm Intelligence. *Proceedings of Seventh Annual Meeting of the Robotics Society of Japan*, pages 425–428, Tokyo, 1989. RSJ Press.
- [21] Hackwood, S., and Beni, G. Self-organizing Sensors by Deterministic Annealing. *Proceedings of IEEE/RSJ International Conference on Intelligent Robot and Systems — IROS'91*, pages 1177–1183, Los Alamitos, 1991. IEEE Computer Society Press.
- [22] Hackwood, S., and Beni, G. Self-organization of Sensors for Swarm Intelligence. *Proceedings of IEEE International Conference on Robotics and Automation*, pages 819–829, Nice, 1992. IEEE Computer Society Press.
- [23] Yim, M. A Reconfigurable Modular Robot with Many Modes of Locomotion. *Proceedings of the 1993 JSME International Conference on Advanced Mechatronics*, pages 283–288, Tokyo, 1993. JSME Press.
- [24] Yim, M. Locomotion with a unit-modular reconfigurable robot. PhD thesis, Department of Mechanical Engineering, Stanford University, Stanford, 1994.
- [25] Yim, M. New Locomotion Gaits. *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 2508–2524, San Diego, 1994. IEEE Computer Society Press.
- [26] Murata, S., Kurokawa, H., and Kokaji, S. Self-Organizing Machine. Video Proceedings, 1995 IEEE International Conference on Robotics and Automation, Nagoya, Japan, May 1995.
- [27] M. Ashley-Rollman, S. Goldstein, P. Lee, T. Mowry, and P. Pillai. *Meld: A declarative approach to programming ensembles*, Proceedings of the IEEE International Conference on Robots and Systems, 2007. pp.2794–2800.
- [28] Charron-Bost, B., Delporte-Gallet, C., and Fauconnier, H. *Local and temporal predicates in distributed systems*, ACM Transactions on Programming Languages and Systems, 17(1):157–179, 1995.
- [29] DeRosa, M., Goldstein, S., Lee, P., Campbell, J., and Pillai, P. *Scalable shape sculpting via hole motion: Motion planning in lattice constrained modular robots*, Proceedings of the IEEE International Conference on Robotics and Automation, 2006. pp.1462–1468.
- [30] Dewey, D., and Srinivasa, S.S. *A planning framework for local metamorphic systems*, Technical Report CMU-RI-TR-XX, The Robotics Institute, Carnegie Mellon University, 2007.
- [31] Ashley-Rollman, M.P., De Rosa, M., Srinivasa, S.S., Pillai, P., Goldstein, S.C., Campbell, J. *Declarative Programming for Modular Robots*, IEEE/RSJ International Conference on Intelligent Robots and Systems, Workshop on Self-Reconfigurable Robots and Systems and Applications, 2007. pp.1–6.
- [32] Pamecha, A., Ebert-Uphoff, I., and Chirikjian, G. Useful metrics for modular robot motion planning. *IEEE Transactions on Robotics and Automation*, 13(4):531–545, 1997.
- [33] Vassilvitskii, S., Kubica, J., Rieffel, E.G., Suh, J.W., and Yim, M. On the General Reconfiguration Problem for Expanding Cube Style Modular Robots. *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*, pages 801–808, Washington, 2002. IEEE Computer Society Press.
- [34] Kurokawa, H., Tomita, K., Kamimura, A., Yoshida, E., Kokaji, S., and Murata, S. Distributed Self-reconfiguration Control of Modular Robot M-TRAN. *Proceedings of the Twenty-Seventh Annual International Conference on Cement Microscopy*, pages 254–259, Victoria, 2005. IEEE Computer Society Press.

- [35] Hosokawa, K., Fujii, T., Kaetsu, H., Asama, H., Kuroda, Y., and Endo, I. Self-organizing collective robots with morphogenesis in a vertical plane. *JSME International Journal Series C Mechanical Systems Machine Elements and Manufacturing*, 42:195–202, 1999.
- [36] Butler, Z., Murata, S., and Rus, D. Distributed Replication Algorithms for Self-Reconfiguring Modular Robots. *Proceedings of 6th International Symposium on Distributed Autonomous Robotic Systems (DARS'02)*, pages 25–27, Fukuda, 2002. <http://groups.csail.mit.edu/drl/wiki/images/c/c5/dars02.pdf>
- [37] Walter, J., Tsai, E., and Amato, N. Algorithms for Fast Concurrent Reconfiguration of Hexagonal Metamorphic Robots. *IEEE Transactions on Robotics*, 21(4):621–631, 2005.
- [38] Ünsal, C., and Khosla, P.K. A Multi-layered Planner for Self-Reconfiguration of a Uniform Group of I-Cube Modules. *IEEE International Conference on Intelligent Robots and Systems. Vol. 1*, pages 598–605, Maui, 2001. IEEE Computer Society Press.
- [39] Reif, J.H., and Slee, S. Optimal Kinodynamic Motion Planning for 2D Reconfiguration of Self-Reconfigurable Robots. Robotics: Science and Systems, Conference, Georgia Institute of Technology, Atlanta, GA, June 27–30, 2007. <http://www.roboticsproceedings.org/rss03/p20.pdf>
- [40] Aloupis, G., Collette, S., Demaine, E.D., Langerman, S., Sacristán, V., and Wührer, S. Reconfiguration of Cube-Style Modular Robots Using  $O(\log n)$  Parallel Moves. In Hong, S.H., Nagamochi, H., Fukunaga, T., editors, *Proceedings of the 19th Annual International Symposium on Algorithms and Computation — ISAAC 2008*, pages 342–353, Gold Coast, 2008. Springer-Verlag.
- [41] Christensen, D., Ostergaard, E., and Lund, H.H. *Metamodule control for the atron self-reconfigurable robotic system*, Proceedings of the The 8th Conference on Intelligent Autonomous Systems, Amsterdam, 2004. pp.685–692.
- [42] Dewey, D., Srinivasa, S.S., Ashley-Rollman, M.P., De Rosa, M., Pillai, P., Mowry, T.C., Campbell, J.D., and Goldstein, S.C. *Generalizing Metamodules to Simplify Planning in Modular Robotic Systems*, Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems, 2008. pp.1338–1345.
- [43] Casal, A., and Yim, M. Self-Reconfiguration Planning For a Class of Modular Robots. *Proceedings of SPIE. Vol. 3839. Sensor Fusion and Decentralized Control in Robotic Systems. Vol. II*, pages 246–257, Boston, 1999. SPIE Press.
- [44] Nelson, C.A. A framework for self-reconfiguration planning for unit-modular robots. Phd Thesis, Purdue University, Department of Mechanical Engineering, Purdue, 2005.
- [45] Gay, S. Roombots: Toward Emancipation of Furniture. A Kinematics-Dependent Reconfiguration Algorithm for Chain-Type Modular Robots. Master Thesis, Ecole Polytechnique, Department of Computer Science, Ecole, 2007.
- [46] Shen, W.-M., Salemi, B., and Will, P. Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots. *IEEE Transactions on Robotics and Automation*, 18(5):700–712, 2002.
- [47] Hou, F., and Shen, W.-M. Distributed, Dynamic, and Autonomous Reconfiguration Planning for Chain-Type Self-Reconfigurable Robots. *Proceedings of 2008 IEEE International Conference on Robotics and Automation*, pages 3135–3140, Pasadena, 2008. IEEE Computer Society Press.
- [48] Hou, F., Shen, W.-M. On the Complexity of Optimal Reconfiguration Planning for Modular Reconfigurable Robots. *Proceedings of 2010 IEEE International Conference on Robotics and Automation*, pages 2791–2796, Anchorage, 2010. IEEE Computer Society Press.
- [49] Cook, S.A. The Complexity of Theorem Proving Procedures. In Harrison, M.A., Banerji, R.B., Ullman, J.D., editors, *STOC '71 Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, 1971. Association for Computing Machinery Press.
- [50] Gu, J., Purdom, P., Franco, J., and Wah, B. Algorithms for the Satisfiability (SAT) Problem: A Survey. In Johnson, David, and Trick, Michael, editors, *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, pages 19–152. American Mathematical Society, Providence, Rhode Island, 1996.
- [51] Fleurent, J. Genetic algorithms and hybrids for graph coloring. *Annals of Operations Research*, 63(3):437–461, 1996.
- [52] Hao, J., and Dorne, R. A new population-based method for satisfiability problems. In Cohn, A.G., editor, *Proceedings of 11th European Conference on Artificial Intelligence*, pages 135–139, Amsterdam, 1994. John Wiley & Sons.
- [53] Jong, K., and Spears, W. Using genetic algorithms to solve np-complete problems. In Schaffer, J.D., editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 124–132, Fairfax, 1989. Morgan Kaufmann Publishers Inc.
- [54] Voorn, R., Dastani, M., and Marchiori, E. Finding simplest pattern structures using genetic programming. In Spector, L., Goodman, E.D., Wu, A., Langdon, W.B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshek, S., Garzon, M.H., Burke, E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 3–10, San Francisco, 2001. Morgan Kaufmann Publishers Inc.
- [55] Hao, J., Lardeux, F., and Saubion, F. A hybrid genetic algorithm for the satisfiability problem. *Proceedings of the 1st International Workshop on Heuristics*, pages 102–109, Beijing, 2002. Springer-Verlag.
- [56] Armando, A., and Giunchiglia, E. Embedding complex decision procedures inside an interactive theorem prover. *Annals of Mathematics and Artificial Intelligence*, 8(3-4):475–502, 1993.
- [57] Giunchiglia, E., and Sebastiani, R. Applying the Davis-Putnam procedure to nonclausal formulas. In Lamma, E., Mello, P., editors, *AI\*IA 99: Advances in Artificial Intelligence, 6th Congress of the Italian Association for Artificial Intelligence*, pages 84–94, Bologna, 2000. Springer-Verlag.
- [58] Kautz, H., Selman, B., and McAllester, D. Exploiting variable dependency in local search. *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 7–9, Nagoya, 1997. Morgan Kaufmann Publishers Inc.
- [59] Muhammad, R., and Stuckey, P.J. A Stochastic Non-CNF SAT Solver. In Yang, Q., Webb, G.I., editors, *PRICAI 2006: Trends in Artificial Intelligence, 9th Pacific Rim International Conference on Artificial Intelligence*, pages 120–129, Guilin, 2006. Springer-Verlag.
- [60] Sebastiani, R. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994.
- [61] Selman, B., Kautz, H., and Cohen, B. Noise strategies for improving local search. In Hayes-Roth, B., Korf, R.E., editors, *AAAI'94 Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 337–343, Seattle, 1994. American Association for Artificial Intelligence.
- [62] Stachniak, Z. Going non-clausal. *5th International Symposium on Theory and Applications of Satisfiability Testing: SAT 2002*, pages 316–322, Cincinnati, 2002. Springer-Verlag.
- [63] Thiffault, C., Bacchus, F., and Walsh, T. Solving non-clausal formulas with DPLL search. In Hoos, H.H., Mitchell, D.G., editors, *Theory and Applications of Satisfiability Testing: 7th International Conference: SAT 2004*, pages 663–678, Vancouver, 2004. Springer-Verlag.
- [64] Bessiere, C., Hebrard, E., and Walsh, T. Local Consistencies in SAT. *Theory and applications of satisfiability testing: SAT 2003: international conference on theory and applications of satisfiability testing N 6*, pages 400–407, Santa Margherita Ligure , 2003. Springer-Verlag.
- [65] Davis, M., Logemann, G., and Loveland, D. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.
- [66] Frisch, A., and Peugniez, T. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search. In Nebel, B., editor, *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 282–288, Seattle, 2001. pp. 282 - 288. Morgan Kaufmann Publishers Inc.
- [67] Frisch, A.M., Peugniez, T.J., Doggett, A.J., and Nightingale, P.W. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. *Journal of Automated Reasoning*, 35(1-3):143–179, 2005.
- [68] Genisson, R., and Jegou, P. Davis and Putnam Were Already Forward Checking. In Wahlster, W., editor, *Twelfth European Conference on Artificial Intelligence*, pages 180–184, Budapest, 1996. John Wiley and Sons.
- [69] Gent, I. Arc Consistency in SAT. In van Harmelen, F., editor, *Proceedings of the Fifteenth European Conference on Artificial Intelligence*, pages 121–125, Lyons, 2002. IOS Press.
- [70] Kasif, S. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *Artificial Intelligence*, 45(3):275–286, 1990.
- [71] Prestwich, S.D. Local search on SAT-encoded colouring problems. *Theory and applications of satisfiability testing: SAT 2003: international conference on theory and applications of satisfiability testing N 6*, pages 105–119, Santa Margherita Ligure , 2003. Springer-Verlag.
- [72] Sabin, D., and Freuder, G. Contradicting Conventional Wisdom in Constraint Satisfaction. In Cohn, A.G., editor, *Proceedings of the Eleventh*

- European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, 1994. John Wiley and Sons.
- [73] Walsh, T. SAT v CSP. In Dechter, R., editor, *Sixth International Conference on Principles and Practice of Constraint Programming*, pages 441–456, Singapore, 2000. Springer-Verlag.
- [74] Iwama, K., and Miyazaki, S. SAR-variable complexity of hard combinatorial problems. *IFIP Transactions A: Computer Science and Technology*, 1:253–258, 1994.
- [75] Büttner, M., and Rintanen, J. Improving parallel planning with constraints on the number of operators. In Biundo, S., Myers, K., and Rajan, K., editors, *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, pages 292–299, Monterey, 2005. AAAI Press.
- [76] Ernst, M., Millstein, T., and Weld, D. Automatic SAT-Compilation of Planning Problems. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Nagoya, 1997. Morgan Kaufmann Publishers Inc.
- [77] Kautz, H. SATPLAN04: Planning as Satisfiability. In Edelkamp, S., Hoffmann, J., Littman, M., and Younes, H., editors, *Proceedings of the 4th International Planning Competition at the 14th International Conference on Automated Planning and Scheduling*, pages 44–45, Whistler, 2004. AAAI Press.
- [78] Kautz, H., McAllester, D., and Selman, B. Encoding Plans in Propositional Logic. In Aiello, L.C., Doyle, J., Shapiro, S.C., editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 374–384, Cambridge, Massachusetts, 1996. Morgan Kaufmann Publishers Inc.
- [79] Kautz, H., and Selman, B. Planning as Satisfiability. In Neumann, B., editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, Vienna, 1992. John Wiley & Sons.
- [80] Kautz, H., and Selman, B. Pushing the envelope: planning, propositional logic, and stochastic search. In Brewka, G., editor, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Annual Conference on Innovative Applications of Artificial Intelligence*, pages 1194–1201, Portland, 1996. AAAI Press.
- [81] Kautz, H., and Selman, B. Unifying SAT-based and graph-based planning. In Dean, T., editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 318–325, Stockholm, 1999. Morgan Kaufmann Publishers Inc.
- [82] Mattmüller, R., and Rintanen, J. Planning for temporally extended goals as propositional satisfiability. In Veloso, M., editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1966–1971, Hyderabad, 2007. AAAI Press.
- [83] Rintanen, J. Compact representation of sets of binary constraints. In Perini, A., Penserini, L., and Peppas, P., editors, *Proceedings of the 17th European Conference on Artificial Intelligence*, pages 143–147, Trento, 2006. IOS Press.
- [84] Rintanen, J. Evaluation strategies for planning as satisfiability. In Lopez de Mantaras, R., and Saitta, L., editors, *ECAI 2004: Proceedings of the 16th European Conference on Artificial Intelligence*, pages 682–687, Valencia, 2004. IOS Press.
- [85] Rintanen, J. Heuristic Planning with SAT: Beyond Uninformed Depth-First Search. In Li, J., editor, *AI 2010: Advances in Artificial Intelligence*, pages 415–424, Adelaide, 2010. Springer-Verlag.
- [86] Rintanen, J. Heuristics for Planning with SAT. In Cohen, D., editor, *Principles and Practice of Constraint Programming: 16th International Conference*, pages 414–428, St. Andrews, 2010. Springer-Verlag.
- [87] Rintanen, J. Planning graphs and propositional clause-learning. In Brewka, G., and Doherty, P., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference*, pages 535–543, Sydney, 2008. AAAI Press.
- [88] Rintanen, J. Symmetry reduction for SAT representations of transition systems. In Giunchiglia, E., Muscettola, N., and Nau, D., editors, *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, pages 32–40, Trento, 2003. AAAI Press.
- [89] Rintanen, J. A planning algorithm not based on directional search. In Cohn, A.G., Schubert, L.K., and Shapiro, S.C., editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference*, pages 617–624, Trento, 1998. Morgan Kaufmann Publishers Inc.
- [90] Rintanen, J. Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formulae. In Nieuwenhuis, R., and Voronkov, A., editors, *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 362–376, Havana, 2001. Springer-Verlag.
- [91] Rintanen, J., Heljanko, K., and Niemelä, I. Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. Technical Report 216, Institute of Computer Science, University of Freiburg, Freiburg, Germany, 2005.
- [92] Rintanen, J., Heljanko, K., and Niemelä, I. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.
- [93] Rintanen, J., Heljanko, K., and Niemelä, I. Parallel encodings of classical planning as satisfiability. In Alferes, J.J., and Leite, J., editors, *Logics in Artificial Intelligence: 9th European Conference*, pages 307–319, Lisbon, 2004. Springer-Verlag.
- [94] Rintanen, J., and Jungholt, H. Numeric state variables in constraint-based planning. In Biundo, S., and Fox, M., editors, *Recent Advances in AI Planning: 5th European Conference on Planning*, pages 109–121, Durham, 2000. Springer-Verlag.
- [95] Wehrle, M., and Rintanen, J. Planning as satisfiability with relaxed  $\exists$ -step plans. In Orgun, M., and Thornton, J., editors, *AI 2007: Advances in Artificial Intelligence: 20th Australian Joint Conference on Artificial Intelligence*, pages 244–253, Surfers Paradise, Gold Coast, Australia, 2007. Springer-Verlag.
- [96] van Gelder, A. Another Look at Graph Coloring via Propositional Satisfiability. In Mehrotra, A., Johnson, D.S., and Trick, M., editors, *Computational Symposium on Graph Coloring and its Generalizations*, pages 48–54, Ithaca, New York, 2002. Springer-Verlag.
- [97] Bouhmala, N., Granmo, O.-C. Stochastic Learning for SAT- Encoded Graph Coloring Problems. *International Journal of Applied Metaheuristic Computing*, 1(3):1–19, 2010.
- [98] Velev, M.N. Exploiting hierarchy and structure to efficiently solve graph coloring as SAT. In Gielen, G., editor, *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 135–142, San Jose, 2007. IEEE Computer Society Press.
- [99] Poljak, S., and Tuza, Z. Maximum cuts and largest bipartite subgraphs. In Cook, William, Lovasz, Laszlo, Seymour, Paul, editors, *Combinatorial Optimization*, pages 181–244. American Mathematical Society, Providence, Rhode Island, 1995.
- [100] Cheriyan, J., Cunningham, W.H., Tuncel, L., and Wang, Y. A linear programming and rounding approach to Max 2-Sat. In Johnson, David, and Trick, Michael, editors, *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, pages 395–414. American Mathematical Society, Providence, Rhode Island, 1996.
- [101] Mahajan, M., and Raman, V. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31(2):335–354, 1999.
- [102] Hoos, H.H. SAT-Encodings, Search Space Structure, and Local Search Performance. In Dean, T., editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 296–302, Stockholm, 1999. Morgan Kaufmann Publishers Inc.
- [103] Plotnikov, A.D. A Logical Model of HCP. *International Journal of Mathematics and Mathematical Sciences*, 26(11), 2001.
- [104] <http://people.cs.ubc.ca/~hoos/SATLIB/index-ubc.html>
- [105] [http://parallelimm.uran.ru/mvc\\_now/hardware/supercomp.htm](http://parallelimm.uran.ru/mvc_now/hardware/supercomp.htm)
- [106] Navarro, J.A., and Voronkov, A. Generation of Hard Non-Clausal Random Satisfiability Problems *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 436–442, Pittsburgh, 2005. AAAI Press.

# Towards a real-time simulation environment on the edge of current trends

Eugene Chemeritskiy

The Faculty of Computational Mathematics and  
Cybernetics, Moscow State University,  
Moscow, Russia  
[tyz@lvk.cs.msu.su](mailto:tyz@lvk.cs.msu.su)

Konstantin Savenkov (advisor)

The Faculty of Computational Mathematics and  
Cybernetics, Moscow State University,  
Moscow, Russia  
[savenkov@cs.msu.su](mailto:savenkov@cs.msu.su)

**This paper is devoted to renewing of the simulation project that has been undoubtedly a successful one and has endured a wide range of tasks, but is slowly and inevitable getting obsolete. In attempt to stay in the top, the development of the new runtime is started taking into account some historical regularities and currents trends in the distributed real-time simulation and some adjoining areas. The paper describes the problem scope resulted from application of the considered technologies, analyzes its possible solutions and estimates the related labor cost.**

*General Terms: Simulation Runtime, Distributed Real-time and Embedded Systems, High Level Architecture.*

## I. INTRODUCTION

In the 1990s the Computer Systems Laboratory (CS Lab) at Computational Mathematics and Cybernetics department of the Moscow State University developed a parallel modeling and simulation system called DYANA [1]. This simulation system has been used a lot as a basis for researches and development of a number of specialized simulation tools. One of these tools called STAND [2] is a hardware/software environment for hardware-in-the-loop simulation of the distributed real-time and embedded systems (DRE).

The STAND environment has been applied to a number of DRE simulation projects and proved its efficiency. To remain at the same high advantageous level in the context of fast progress in the whole IT area, it was decided to construct new runtime following the current trends to standardization in the simulation fields.

The standard-compliant runtime subsystem automatically guarantees model compatibility. Models written in accordance with the standard specifications could be always executed with use of this runtime. Similarly, natively developed models could be executed by any other certified system. This compatibility could result in product popularization and the formation of user community, and a large number of users, in its turn, could accelerate the project development and lead to its further improvement.

Replacement of the STAND native runtime raises a number of problems that could be separated into the following groups in accordance with their nature.

### A. *Designing of DRE-supporting runtime in pursuance of the latest simulation trends*

Being quite a specific simulation case, DRE simulation imposes some additional requirements to the runtime. Currently, there is no any off-the-rack and well-fitted simulation standard. Thereby some adjoining simulation areas have been explored. In attempt to mark the current trends in these areas, the third section of this paper gives a brief concept of the simulation historical path and its progress regularities. For each of the adverted innovations, the application goals and prospects are described in context of the considered project development.

Once the runtime is conceptually designed, the time comes to its implementation. Despite the considered technologies are relatively new, all of them have certain users and it is possible to learn from their experience. The refinement of the existing solutions and their adaptation to the purposes of the considered project is far less labor-intensive than the development from scratch. Thereby, the paper describes some possibilities for the adoption of the turnkey solutions.

### B. *Integration to the STAND environment and maintenance of the legacy projects*

The next aspect of STAND runtime replacement is reuse of the other STAND components. STAND software package includes a number of additional assistance subsystems such as trace collector, dynamic visualizer, version control system, integrated model development environment and so on. All listed subsystems are interconnected and have certain dependencies from each other. Due to the runtime is not a rule exception, replacement of this subsystem generates a large amount of integration problems.

The integration problems are compounded by the necessity of legacy project maintenance. The STAND environment provides a highly specialized C-based model development language. This language includes some functionality to

---

This work was supported in part by the Ministry of education and science of the Russian Federation under Grant "Development of an integrated environment and complex analysis methods for distributed real-time computer systems functioning".



simplify DRE simulation (e.g. integrated support of the DRE data transmitting channels). These features are often implemented as low-level functions integrated deeply inside the runtime. Because of the interface limitations imposed to the new simulation runtime by specifications of the selected simulation standard, the effective implementation of the mentioned functionality becomes a serious research challenge.

## II. THE STAND SIMULATION ENVIRONMENT

Modern DRE systems consist of multiple devices connected by data transfer network which contain dozens of channels. Development of DRE devices and of the DRE itself is a distributed process performed by several workgroups and the device prototypes become ready for integration in different points of time. To meet the deadlines for DRE development, the integration testing operations should begin in advance, when some of the components are not implemented yet [2].

STAND enables incremental DRE gradual integration the DRE according to the schedule of incoming devices. On early stages of the DRE integration, most (or all) of the devices are represented by the simplest simulation models reflecting only a basic schedule of data exchanges. Then the detail level is gradually increased upto full-scale models that include software of real devices and generate appropriate data matching the one generated by the device prototypes. On the next step of integration the models are step-by-step replaced by real devices that perform data exchange through the real channels.

On every listed stage of the DRE integration, the available set of devices and models could be analyzed and validated. This approach provides the abilities to detect and fix existing device errors in the earliest development phases and to reduce the DRE development cost subsequently.

The considered simulation environment contains tools intended to solve the following simulation-related tasks [2]:

1. Development of simulation models of DRE devices and auxiliary synthetic simulation models (e.g. model of the external environment);
2. Support for real-time execution of the available DRE component set including the model-device interactions through hardware channels;
3. Dynamic visualization of the simulation state and results in graphical and tabulated form and abilities for human-assisted control of the simulation;
4. Recording and processing of the simulation results, interaction with hardware monitors for data exchange channels.

## III. TRACING CURRENT TRENDS

### A. Interface standardization

Simulation as a method for exploration of diverse object properties and regularities among them outruns the advent of computers for many years. However, its rapid development started after the complex mathematical calculations had been assigned to fast and reliable computers. In the beginning of the

1950s, the term simulation acquired the default meaning of digital computer simulation. Subsequently the simulation was defined as a combination of designing of the observed system model and holding the necessary experiment set on digital computers [3].

The observed system here means a separated part of the world corresponding to the domain of researcher interests. This world view is isolated during the experiment and consists of a component set. Each of these components is characterized by its property set and the dynamics of their change. Such a system could exist in reality or be imagined, can receive information and/or transmit it to its environment [4].

Abstraction that holds a subset of the observed system properties is called a model. The selected property subset should meet the objectives of the simulation. The result of simulation has any sense only in case of the simulation goals were properly identified and the constructed model is adequate to these goals [4].

From the very beginning of the simulation history the observed systems always tended to be represented in deeper detail level. This tension results in the increasing size and complexity of developed simulation model. This growth required a respective performance increase from computer systems, and this fact resulted in emergence of parallel simulation systems. These systems share the simulation task across multiple computing nodes. Typically such systems were implemented locally within the organization that wanted to use it (in accordance with this classification STAND is a parallel system created in the CS Lab) [5].

The complexity of the models was not the only factor leading to computer simulation tool evolution. The scope of simulation has been growing either. After new simulation problem types appeared, the related requirements were imposed to modeling and simulation tools. For instance, distributed simulation is often required in case of joint product development when different product component are produced by a number of workgroups located in different organizations. This type of simulation intends encompassing of several geographically separated simulation systems, which in turn may consist of a single compute node, or be a parallel system. Historically, the appearance of this task type led to the creation of distributed simulation systems that provide an essential set of services to the simulation participants and ensure its consistent behavior [5].

The next and the latest commonly recognized step in the modeling and simulation tool evolution is a standardizing of the distributed system interfaces. Using of this principle results in possibility to combine among a variety of independent simulation systems and create a general model that can be handled by every distributed system corresponding to the standard specifications [6].

### B. DRE simulation specific

The above classification groups existing simulation tasks and tools according to node configuration of the underlying computer system. There are lots of other features that could serve as a classification criterion. The one that is important in

context of this paper is a range of supported participant types: syntactic (could be completely represented by its model) or live participants (represented by external entities). Generally live simulation type is further separated into human-in-the-loop and hardware-in-the-loop simulation depending whether the experiments requires the human presence or the external entity is a fully automated one.

Hardware-in-the-loop simulation often includes a number of physical devices, which require their data to be delivered with the respect to a given period of time (deadline), as the participants. A meeting of the deadlines in such systems is a focus of the of real-time system problematic, which are defined as those systems in which a correctness of the system depends not only on the logical results of computation, but also on the time at which these results are produced. Thereby model time must be synchronized with the astronomical one when the model interacts with hardware.

A real-time application is usually comprised of a set of cooperating tasks and they need a reliable prediction of the worst-case scenario. Apart from satisfying the timing constraints, another important characteristic of real-time systems is the notion of predictability.

Real-time systems are usually classified into two categories based on the nature of deadline, namely, hard real-time systems, in which the consequences of deadline breaking may be catastrophic and soft real-time systems, in which the utility of results produced by a task with a soft deadline decreases over time after the deadline expires. Examples of hard real-time systems are avionic control and nuclear plant control. Telephone switching system and video streaming applications are examples for soft real-time systems [6].

Besides the support of hard and soft real-time simulation, the simulation system intended to be used in DRE development should interact with additional tools providing the following capabilities:

1. Verification of the DRE devices compliance to the technical specification;
2. Integrated testing and debugging of distributed DRE software;
3. Performance and robustness evaluation of the DRE architecture;
4. Scheduling of data transfers and validation of the constructed schedules.

#### IV. DESIGNING THE RUNTIME

The High Level Architecture (HLA) is the conventional standard in the field of distributed simulation and de facto is supported by the most of non-distributed simulation tools and by the community of distributed model developers. This standard is acceptable for DRE simulation, so it was chosen as a base standard.

Despite its initial focus on distributed simulation, using the HLA standard results in some benefits in case of the parallel simulation system (the nodes are located closely) development either. The system based on this standard can become a

member of the distributed simulation and supports a range of polytypic simulation models (e.g. as-fast-as-possible synthetic models and any other types supported by the HLA standard) out of the box. In addition, the operational power of utilities devoted to distributed simulation enables easy setup of parallel simulation system node set.

The HLA standard does not currently address real-time simulation and HLA compliant simulation could not require any Quality of Service (QoS) from the underlying middleware (RTI). Indeed, there are several problems that should be solved to enable it [8]:

1. No interfaces provided to specify end to end prediction requirements for federate;
2. Management of underlying operating system(s) is unavailable;
3. In distributed case, HLA supports two transportation types only: the reliable one and the best-effort one (usually encoded with the TCP and UDP network protocols) which are not suitable for real-time constraints.

These different limitations have crucial impact for real-time simulation systems where the amount and predictability of RTI overhead is an important design factor. Thereby the considered project requires development of an additional data transmitting layer with a real-time support. Fortunately, there exist a number of related standards and associated implementations. One of the most widespread standards in this domain is the OMG Data Distribution Service (DDS) [9].

The DDS standard defines a large number of QoS policies for inter-process connection. Considering the need to meet the constraints of real time, the represented project implementation should follow the HLA standard specifications in context of inter-process communication semantics and be based on DDS standard in context of data transmission protocols.

To summarize the above, the new simulation runtime is conceptually formed around the HLA simulation standard. Because of the DRE simulation requires from the runtime some extra features (such as QoS enabled connections) not specified by HLA, the additional data transmitting middleware level (specified by the DDS standard) should underlay the usual HLA middleware (RTI) and possibly extend its functionality. STAND consists of a number of computational nodes and this imposes the resulting combined middleware to be deployed on each of them.

##### A. *The High Level Architecture standard*

The roots for the HLA stem from distributed virtual environments into which users, possibly at geographically distant locations, can be encompassed. The HLA standard is a conceptual heir of Distributed Interactive Simulation (DIS) [10], which is a highly specialized simulation standard in the domain of training environments, and is used mostly for military purposes. The primary mission of DIS is to enable interoperability among separated modeling and simulation systems and to allow the joint simulation with the merged systems participation.

HLA standard remains the DIS principle relevant and extends it to the idea of polytypic model merging. Thus the HLA development began in 1993 when the Defense Advanced Research Projects Agency (DARPA) designated an award for developing of an architecture that could combine all existing modeling and simulation system types into one federation providing the reuse of existing models and simulation utilities.

There are several federation types (so called proto-federations) in accordance to the encompassed participant set [11]:

1. The Platform federation type includes DIS-style training simulations (that is real-time human-in-the-loop training simulations);
2. The Joint Training federation type stands for as-fast-as-possible time-driven and event-driven simulation (e.g. command-level military trainings);
3. The Analysis federation includes as-fast-as-possible event-driven simulations such as those that might be used in acquisition decisions;
4. The Engineering federation including hardware-in-the-loop simulations with hard real-time constraints.

The standard already has a pretty reach history and several HLA versions have been published since its appearance. Most of commercial tools currently support HLA version 1516-2000 specification. Some long term projects have being developed less intensively since of their appearance before this version have been published and are still specialized in DMSO 1.3 version. The most advanced tools are compatible with the latest IEEE 1516-2010 (Evolved).

Middleware in computing terms is used to describe a software agent acting as an intermediary between different distributed processes. It is connectivity software which allows, usually, several applications to run on one or several computational nodes and to interact across a network [6].

The middleware involved in HLA is named the Run Time Infrastructure (RTI). The RTI is the software implementation of the HLA Interface Specification. It is a middleware for the proper functioning of distributed simulation in accordance with the principles and specifications from HLA standard [11].

#### B. The Data Distribution Service standard

OMG DDS specifications set the standard of inter-process communication, which is applicable to a broad class of distributed real-time and embedded systems (DRE). The basis of DDS is a data-centric model with the publisher-subscriber architecture (DCPS). The DCPS model forms layer, which allows the integrated processes to set a typed shared data or get the latest its version. As parts of DCPS, the global data space and namespace are created. The publisher process (the one who wants to create a shared object) should make the appropriate entries in the global data and name spaces. Similarly, the subscriber process can find the proper objects in the global namespace and access to relevant data. It is important that the announcement of the need to use the shared data and its direct use are time separated, and this approach enables the quality of service connection [7].

TABLE I  
RTI IMPLEMENTATIONS

RTI	Developer	License type
ARTIS GAIA	University of Bologna	Open Source <sup>1</sup>
CERTI	ONERA	GPL v2 or later
EODiSP	P&P Software	GPL <sup>2</sup>
MAK	MAK Technologies	Commercial
NCWare	Nextel	Commercial
Portico	Portico	CDDL <sup>3</sup>
pRTI	Pitch Technologies	Commercial
RTI NG	Raytheon	Commercial

<sup>1</sup>Full license text is available <http://pads.cs.unibo.it/>

<sup>2</sup>General Public License

<sup>3</sup>Common Development and Distribution License

#### C. Evaluating of a suitable turnkey RTI implementation

There are a lot of off-the-rack RTI implementations (Table I) and this fact gives a hope to get some developments from other projects, learning from their mistakes. Thereby, it was decided to explore the area in more details. The study was conducted among the tools, satisfying (at least partially) to the following criteria:

1. The description of the architecture and principles of implementation are available;
2. The source code of the product is available.
3. The product continues to maintain and develop;
4. The implementation is used for real-time simulation;
5. The implementation is based on the DDS standard;

Most of the examined tools are commercial, and their source code is unavailable. Thereby, benefits from the use of these implementations, taken by the developers of the target simulation system, are limited to the theoretical base. For example it is known that NCWare implementation conforms to DDS standard, and this scheme corresponds to the architectural ideas founded into the basis of considered project. The study found a number of open source systems also, and it was decided to build the target simulation system on the basis of the most suitable of them.

Unfortunately, all of the listed systems have a certain drawbacks in accordance with the purposes of the submitted project. The ARTIS GAIA implementation attracts by its advanced load balancing mechanism supplementation, but the license for this product does not allow the free use of its source code (although it is stated that the project will be fully open in future) [12]. The open source project EODiSP stopped the development in 2006 [13]. Accordingly, there is no one to assist in solving of possible development difficulties encountered. Portico project RTI is implemented using Java and, due to the language specific, it is badly compatible with the real-time simulation that is a primary goal of considered project.

Thereby the best base RTI realization for the development of the considered simulation system a priori is the CERTI one. CERTI is distributed under the GPL license, continues to

evolve, and is implemented in C++ (a number of extra bindings including Java, Python, Fortran and even MATLAB is currently available). In addition, CERTI could be deployed on several combinations of platforms (Windows and Linux, Solaris, FreeBSD...) and compilers (gcc, MSVS, Sun Studio, MinGW...).

#### D. CERTI

For years, the French Aerospace Laboratory (ONERA) develops its own HLA compliant RTI called CERTI. The project started in 1996 and its primary research objective was the distributed simulation itself whereas the appeared HLA standard was the project experiment field. CERTI started with the implementation of the small subset of RTI services, and was used to solve the concrete applications of distributed simulation theory [6].

Since the CERTI project was open sourced in 2002, a large distributed simulation developer community has been formed around the project. In many ways due to contributions of enthusiasts, the CERTI project has grown from basic RTI into a toolset including a number of additional software components that may be useful to potential HLA users.

The CERTI project has always served a base for researches in the domain of distributed simulation, and a number of innovative ideas have been implemented with its use. Thus, the problem of confidential data leak was solved in context of CERTI RTI architecture, and the considered RTI guarantees secure interoperation of simulations belonging to various mutually suspicious organizations [14]. The certain interest for the considered project is a couple of application devoted to high performance and hard real-time simulation.

In spite of HLA is initially designed to support fully distributed simulation applications, it provides a framework for composing not necessarily distributed simulations. Thereby there was created an optimized version of CERTI devoted to simulation deployed on the same shared memory platform and composed simulation running on high-performance clusters [15].

Some experience could also be adopted from ONERA project on simulation of satellite spatial system. Each federate in this federation is a time-stepped driven one. It imposes an additional requirement of hard real-time: the simulation system should meet the deadlines of each step and synchronize the different steps of the different federates [16].

Despite the distribution of commercial products, the project development is still continuing in accordance with the HLA simulation standard progress. Thus, CERTI supports HLA IEEE 1516-2000 version since 2010 in addition to previous DMSO 1.3 version.

#### V. WORK SCOPE ANALYSIS

During the searches of the turnkey projects, the well suitable open source RTI implementation (CERTI) was found. To meet the real-time system requirement the internal of this middleware should gain the property of predictability and an acceptable performance.

The first problem could be solved by RTI refining in according with DDS specifications. During the constructing of the considered RTI to the DDS middleware, it is important to remain the ability of usual distributed simulation. The possible solutions are to implement the optional real-time support or provide the usual RTI-internal interface to the external simulation participants whereas staying the real-time simulator inside.

Test results show that the selected RTI loses to its commercial analogues [17], and this is largely resulted from its centralized architecture. Also the centralized architecture could be a barrier during the refinement related to DDS-compliance. Devoid of the central component the federated architecture seems to be more suitable one. However, the best suitable architecture should be identified in a separate study. Nevertheless, the architectural changes are necessary and justified. Fortunately, the CERTI RTI has been already served as a basis for creating a hard real-time simulator and some experience could be learned from that project.

There is an extra problem caused by high specialization of the STAND runtime. In some cases the functionality of the current STAND runtime could not be simulated even by the combined HLA&DDS middleware, thereby it should be injected into the middleware. Integrated support of physical data transmitting channels is a good example of this case.

Besides the mere building of a new runtime, its replacement results into a number of integration problems related to other components of the STAND environment. Each subsystem provides a certain interface to the others whereas the HLA-compliant runtime has completely different interface set. The best and the easiest way to solve the incompatibility problem is a development of appropriate interface wrappers.

The history of highly specialized standards (and HLA in particular) demonstrates a certain interface steadiness. Even if the interface has changed, the modifications are usually related to the service names and signatures whereas their semantic is the same. This solution provides an ability to replace the runtime again to the better HLA-compliant one or disintegrate some other STAND subsystems into runtime independent separated projects.

Unfortunately, there are some peculiar cases that could not be solved in the described manner. These cases require an embedding the additional hooks into RTI and lead to partial loss of benefits considered above.

The problem of the legacy project maintenance can also be considered as an integration problem, but it deserves more detailed consideration. HLA defines a set of common service devoted to a wide range of simulation tasks. This service set is redundant and inconvenient to be used with the usual DRE simulations whereas the absence of functionalities that could be useful in this particular task.

Despite all the reasoning related to common integration problems remains relevant, the legacy project maintenance problem could be solved with use of another principle. There could be some STAND subsystems requiring renewal, besides its runtime. Thereby if there exists any programming language which is acceptable to DRE simulation, there is a sense in

constructing the HLA-compliant binding for this language and develop an additional translator for old projects.

In summary, the replacement of the current STAND runtime with the concept designed reduces to the following problem set:

1. Replacement of RTI architecture with more complex and productive one;
2. Development of the acceptable interface wrappers for other subsystems;
3. Injecting of some additional functionality and required low-level services.

## VI. CONCLUSION

Using of the HLA distributed simulation standard for building DRE simulation systems gives a certain benefits to the developers, namely, automatic ability to execute any HLA-compliant models and to participate in distributed simulation. Building of the DRE simulation runtime raises a number of development problems. The specific of DRE simulation imposes some additional requirement to the runtime, and specifications of the HLA standard do not satisfy the appropriate product. There are two possible solutions: addition of QoS policies to existing CERTI implementation and using of the HLA standard over the DDS standard. The considered solutions were analyzed and a second one was chosen.

Due to existence of some more or less suitable turnkey RTI, the upcoming development promises to be easier than the development from scratch. It reduces to refinement of the RTI architecture, injecting the RTI with some new functionality and developing of interface wrappers.

## REFERENCES

- [1] R.L. Smeliansky, A.G. Bakhmurov, and V.A. Kostenko, "DYANA - an environment for simulation and analysis of distributed multiprocessor computer systems," Moscow State University, Computational Math. and Cybern. Dept., 1999.
- [2] V.V. Balashov et al., "A hardware-in-the-loop simulation environment for real-time systems development and architecture evaluation," in International Conference on Dependability of Computer Systems, 2008, pp. 80-86.
- [3] R.G. Sargent, "Requirements of a modeling paradigm," in Winter Simulation Conference, Arlington, USA, 1992, pp. 780- 782.
- [4] E.H. Page, "Simulation modeling methodology: principles and etiology of decision support," Department of Computer Science, Virginia Tech, Blacksburg, USA, Ph.D. Dissertation 1994.
- [5] R.E. Nance, A history of discrete event simulation programming languages. Blacksburg, USA, 1993.
- [6] E. Noulard, J.Y. Rousselot, and P. Siron, "Spring Simulation Interoperability Workshop," in CERTI, an open source RTI, why and how, San Diego, USA, 2009.
- [7] Object Management Group; Object Interface Systems, Inc; Real-Time Innovations, Inc; THALES, Data Distribution Service for Real-time Systems, version 1.2., 2007.
- [8] M. Adelantado, P. Siron, and Chaudron J.B., "Towards an HLA runtime infrastructure with hard real-time capabilities," in International Simulation Multi-Conference, Ottawa, Canada, 2010.
- [9] Real-Time Innovations, Inc. (RTI), "OMG Data-Distribution Service (DDS): architectural overview," 2004.
- [10] Richard D. Fujimoto, Parallel and distributed simulation systems, 2000.
- [11] IEEE Std 1516.1-2000, "IEEE standard for modeling and simulation (M&S) High Level Architecture (HLA) - federate Interface specification," 2001.
- [12] L. Bononi, M. Bracuto, D'Angelo G., and Donatiello L., "A new adaptive middleware for parallel and distributed Simulation of dynamically interacting systems," in Distributed Simulation and Real-Time Applications, 2004, pp. 178 - 187.
- [13] I. Birrer, B. Carnicero-Dominguez, M. Egli, B. Carnicero-Dominguez, and A. Pasetti, "EODiSP - an open and distributed simulation platform," in International Workshop on Simulation for European Space Programmes, Noordwijk, the Netherlands, 2006.
- [14] P. Bieber, D. Raujol, and P. Siron, "Security architecture for federated cooperative information systems," in Annual Computer Security Applications Conference, New Orleans, USA, 2000.
- [15] M. Adelantado, J.L. Bussenot, J.Y. Rousselot, P. Siron, and Betoule M., "HP-CERTI: towards a high performance, high availability open source RTI for composable simulations," in Fall simulation interoperability workshop, Orlando, USA, 2004.
- [16] B. d'Ausbourg, P. Siron, and E. Noulard, "Running real time distributed simulations under Linux and CERTI," in European Simulation Interoperability Workshop, Edimburgh, Scotland, 2008.
- [17] L. Malinga and WH. Le Roux, "HLA RTI performance evaluation," in European Simulation Interoperability Workshop, Istanbul, Turkey, 2009, pp. 1-6.

# The problem of placement of visual landmarks

Anna Gorbenko  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: gorbenko.aa@gmail.com

Maxim Mornev  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: max.mornev@gmail.com

Vladimir Popov  
Department of Mathematics and  
Mechanics  
Ural State University  
Ekaterinburg, Russia, 620083  
Email: Vladimir.Popov@usu.ru

**Abstract**—Many robotic problems are computationally hard. Implementation of various heuristics for solving such problems greatly complicates the development of efficient software for robotic systems. In recent years among developers of robotic software formed a direction of the development of individual solvers. Such solvers are designed for specific hard problems. It should be noted that developed specialized programming languages for robotic logic. These languages allow efficient scale logic circuits produced by the solver for large robotic complexes. On this basis it can be argued that now the main problem in this area consists in the developing efficient solvers themselves. In this paper we consider an approach to design of an efficient solver for the problem of placement of visual landmarks. In particular, we present a formalization of the problem of placement of visual landmarks for navigation of mobile robots. We show that this problem is NP-complete. Also we propose an approach to solve this problem. Our approach based on the construction of a logical model. In our construction we give an explicit polynomial reduction from the problem of placing of visual landmarks to the maximum satisfiability problem. Such approach provides effective solution of the considered problem using a high performance computing.

## I. INTRODUCTION

Many robotic problems are computationally hard. In particular, we can mention planning problems, pattern recognition, pattern matching, localization problems, mapping problems, SLAM (simultaneous localization and mapping) and many others. Implementation of various heuristics for solving such problems greatly complicates the development of efficient software for robotic systems. In recent years among developers of robotic software formed a direction of the development of individual solvers. Such solvers are designed for specific hard problems [1] – [4]. It should be noted that developed specialized programming languages for robotic logic [5], [6]. These languages allow efficient scale logic circuits produced by the solver for large robotic complexes. On this basis it can be argued that now the main problem in this area consists in the developing efficient solvers themselves. In this paper we consider an approach to design of an efficient solver for the problem of placement of visual landmarks.

Visual navigation is extensively used in contemporary robotics (see e.g. [7] – [11]). In many cases methods of the visual navigation based on some algorithms of the selection of visual landmarks. Note that even for other approaches (e.g. reactive motion (see, in particular, [12] – [14]) or topological

navigation [15] – [19]) appliance of visual landmarks as an additional method allow significantly increase performance of a navigation system. We can use not only artificial landmarks [20] – [22] but also different objects from environment (see e.g. [23] – [30]).

Navigation systems based on landmarks are most simple and efficient method of orientation. An appliance of artificial landmarks provides a reliable way of autonomous operations. But for appliance of artificial landmarks we need either initial equipment of coverage area of the robot or the robot itself should have a function of self-installation of landmarks. In both cases decreasing of the number of landmarks have significantly influences on improvement of the robot performance. Another way consist in an implementation of natural landmarks. It is allows to avoid a cost of the landmarks installation. But in this case we need essentially more effective methods of visual information processing than for artificial marks (see e.g. [31], [32]). In particular, for search even a simple regularity we need to solve some NP-hard problem (see e.g. [33], [34]). When we do not have sufficiently fast algorithms for solving such problems, we use some self-learning navigation systems (see e.g. [25]). This leads to fast growth of landmarks data base and, finally, to dramatic drop of performance of on-board computer systems. Therefore, problem of minimizing of interest for both artificial and natural landmarks. In the first case we need to minimize the cost of installation of landmarks. In the second case we minimize an expense of computational resources for the new landmarks search and for the identification of available landmarks. A question about a minimization of the number of used landmarks can be formalized as some algorithmic problem where we need to find an explicit placement of landmarks. This placement must safeguard a navigation of mobile objects in a specified area.

## II. PROBLEM DEFINITION

Usually, a problem of placement of visual landmarks in a three dimensional space can be reduced to such problem in a two dimensional space. Frequently, an appropriate reduction can be obtained using the fact that range of heights is too small in compare with horizontal coordinates (an underwater navigation) or by a sampling with a large step (an aerospace navigation). In both cases we can place a set of points, which corresponds to different heights in the one cell of

sampling. When the range of heights is too large and horizontal coordinates have a small variation (a navigation inside a skyscraper), usually, space is divided on horizontal layers lying in one plane. Thus, without loss of generality, we can consider a problem of placement of visual landmarks in the discrete space  $Z^2$ , where  $Z$  is the set of integers. Also we identify every element  $x$  of  $Z^2$  with a square with sides equal to one and the center in  $x$ . A set of points in  $Z^2$  which of interest to navigation we denote by  $N$ . Note that  $N$  is not necessarily a connected area. For instance, we can be interested only in surface facilities and a part of the surrounding area can be covered by water. Let  $S$  be a set of points in  $Z^2$  which permissible to placement of landmarks. It is natural to assume that we are dealing with some limited region  $R$  such that  $N \subseteq R$ ,  $S \subseteq R$ . Since the deployment region  $R$  can contain obstacles or visual landmarks can be visible not from all points of space, it is natural to assign for each point of the set  $S$  its own field of vision which is defined by the function

$$F : S \rightarrow 2^R.$$

We can suppose that  $F$  is given by the sequence of pairs consisting of elements of  $S$  and corresponding subsets. We also consider some constant  $d$  which determines a minimal number of necessary landmarks. Note that the value of  $d$ , usually, does not exceed 4. In the form of a satisfiability problem the considering problem can be formulated as follows:

THE PROBLEM OF PLACEMENT OF VISUAL LANDMARKS (VL):

INSTANCE: A finite set  $R$ ,  $S \subseteq R$ ,  $N \subseteq R$ ,  $F : S \rightarrow 2^R$ , a natural parameter  $k$ , and a natural constant  $d$ .

QUESTION: Is there  $T \subseteq S$  such that  $|T| \leq k$  and for all  $y \in N$  there is  $D \subseteq T$  such that

$$y \in F(x) \text{ for all } x \in D;$$

$$|D| \geq d?$$

### III. COMPLEXITY OF VL

**Theorem.** VL is NP-complete.

**Proof.** Note that  $T \subseteq S$ . Therefore, the number of elements in  $T$  is limited by the number of elements in  $S$ . A size of  $F(x)$  for  $x \in T$  is limited by the size of  $N$ . It is evident that the value

$$\cup_{x \in T} F(x)$$

is computable in polynomial time from the number of elements in  $T$  and the size of  $F(x)$ . So, for the problem VL there is a polynomial algorithm of checking. Therefore, VL is in NP.

Now to prove the theorem it is sufficient to show the hardness of VL. We reduce the 3-set exact cover problem to VL. The 3-set exact cover problem is a well known NP-hard problem [35]. Initial data of this problem is the set

$$U = \{1, 2, \dots, n\}$$

and the set

$$\Sigma = \{X_i \mid 1 \leq i \leq r, X_i \subset U, |X_i| = 3\}.$$

In this problem we need to find out whether there is  $\Pi \subseteq \Sigma$  such that for all  $X, Y \in \Pi$  we have a following

$$X \cap Y = \emptyset,$$

$$\cup_{X \in \Pi} X = U.$$

When  $n$  is not divisible by three, the answer is trivially negative. So, without loss of generality, in the 3-set exact cover problem we can assume that  $n$  is divisible by three.

$$\text{Let } d = 1, k = \frac{n}{3},$$

$$N = \{((n+r-1)(2(r-1)+2i-1), 0) \mid i \in U\},$$

$$S = \{((n+r-1)(n+r-1+2(j-1)),$$

$$(n+r-1)(n+r-1)) \mid 1 \leq j \leq r\}.$$

For any point

$$S_j = ((n+r-1)(n+r-1+2(j-1)), (n+r-1)(n+r-1))$$

from  $S$  consider the triangle  $\mathcal{T}_j$  with vertices in this point and points

$$A_j = (2(n+r-1)(j-1), 0)$$

and

$$B_j = (2(n+r-1)(n+r+j-2), 0).$$

Suppose that for any  $m$ ,  $1 \leq m \leq n+r-1$ , the segment with vertices

$$(2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2(m-1),$$

$$(n+r-1)(n+r-2))$$

and

$$(2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2m,$$

$$(n+r-1)(n+r-2)),$$

represents a tight space of zero width if and only if  $m-r+j \notin X_j$ . Assume that the signal propagates rectilinear and define  $F(S_j)$  as a set of points  $x$  of triangle  $\mathcal{T}_j$  such that  $x$  is visible from  $S_j$ . As  $R$  we consider rectangle, the bottom left corner of which is located at the point  $(0, 0)$  and a upper right corner of which is located at the point

$$(2(n+r-1)(n+3r-3), 2(n+r-1)(n+r-1)).$$

It is easy to see that values of  $R$ ,  $S$ ,  $N$ ,  $F$ ,  $k$ ,  $d$  is defined correctly and their size polynomially depends from the initial data. Let  $\mathcal{T}'_j$  be a triangle with vertices  $S_j$ ,

$$A'_j = (2(n+r-1)(j-1) + (n+r-1)(n+r-2),$$

$$(n+r-1)(n+r-2)),$$

and

$$B'_j = (2(n+r-1)(j-1) + (n+r-1)(n+r),$$

$$(n+r-1)(n+r-2)).$$

Note that

$$A_j \vec{S}_j = ((n+r-1)(n+r-1+2(j-1)) -$$

$$\begin{aligned}
& 2(n+r-1)(j-1), (n+r-1)(n+r-1)) = \\
& ((n+r-1)(n+r-1), (n+r-1)(n+r-1)), \\
& A'_j \vec{S}_j = ((n+r-1)(n+r-1+2(j-1)) - \\
& 2(n+r-1)(j-1) - (n+r-1)(n+r-2), \\
& (n+r-1)(n+r-1) - (n+r-1)(n+r-2)) = \\
& (n+r-1, n+r-1).
\end{aligned}$$

So,  $A'_j \vec{S}_j = (n+r-1)A'_j \vec{S}_j$ . Therefore,

$$A'_j \vec{S}_j \parallel A'_j \vec{S}_j.$$

In this case  $A'_j \in A_j S_j$ . Since

$$\begin{aligned}
& B'_j \vec{S}_j = ((n+r-1)(n+r-1+2(j-1)) - \\
& 2(n+r-1)(n+r+j-2), (n+r-1)(n+r-1)) = \\
& (-(n+r-1)(n+r-1), (n+r-1)(n+r-1)), \\
& B'_j \vec{S}_j = ((n+r-1)(n+r-1+2(j-1)) - \\
& 2(n+r-1)(j-1) - (n+r-1)(n+r), \\
& (n+r-1)(n+r-1) - (n+r-1)(n+r-2)) = \\
& (-n-r+1, n+r-1),
\end{aligned}$$

it is easy to see that  $B'_j \in B_j S_j$ . Clearly, in this case

$$A'_j B'_j \parallel A'_j B'_j.$$

Thus, triangles  $\mathcal{T}_j$  and  $\mathcal{T}'_j$  are similar with the similarity ratio

$$n+r-1.$$

It is easy to see that for all  $j$  segments with vertices

$$\begin{aligned}
& (2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2(m-1), \\
& (n+r-1)(n+r-2))
\end{aligned}$$

and

$$\begin{aligned}
& (2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2m, \\
& (n+r-1)(n+r-2)),
\end{aligned}$$

$1 \leq m \leq n+r-1$ , give us a partition of the segment  $A'_j B'_j$  into  $n+r-1$  equal parts. It is evident that  $A'_{j+1} = B'_j$  where  $j < r$ ,

$$\begin{aligned}
A'_j &= (2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2(m-1), \\
& (n+r-1)(n+r-2))
\end{aligned}$$

where  $m = 1$  and

$$\begin{aligned}
B'_j &= (2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2m, \\
& (n+r-1)(n+r-2))
\end{aligned}$$

where  $m = n+r-1$ . From this and from the similarity of triangles  $\mathcal{T}_j$  and  $\mathcal{T}'_j$  we obtain that points of the segments with vertices

$$(2(n+r-1)(j-1) + 2(m-1)(n+r-1), 0)$$

and

$$(2(n+r-1)(j-1) + 2m(n+r-1), 0)$$

is visible from the point  $S_j$  if and only if points of the segments with vertices

$$\begin{aligned}
& (2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2(m-1), \\
& (n+r-1)(n+r-2))
\end{aligned}$$

and

$$\begin{aligned}
& (2(n+r-1)(j-1) + (n+r-1)(n+r-2) + 2m, \\
& (n+r-1)(n+r-2))
\end{aligned}$$

not form an obstacle. Therefore, the segments with vertices

$$(2(n+r-1)(j-1) + 2(m-1)(n+r-1), 0)$$

and

$$(2(n+r-1)(j-1) + 2m(n+r-1), 0),$$

is visible from the point  $S_j$  if and only if

$$m-r+j \in X_j.$$

Consider the point

$$N_i = ((n+r-1)(2(r-1) + 2i-1), 0).$$

It is easy to see that the point  $N_i$  belongs to the segment

$$\begin{aligned}
& [(2(n+r-1)(j-1) + 2(m-1)(n+r-1), 0); \\
& (2(n+r-1)(j-1) + 2m(n+r-1), 0)]
\end{aligned}$$

if and only if

$$\begin{aligned}
& 2(n+r-1)(j-1) + 2(m-1)(n+r-1) \leq \\
& (n+r-1)(2(r-1) + 2i-1) \leq \\
& 2(n+r-1)(j-1) + 2m(n+r-1).
\end{aligned}$$

Hence,

$$\begin{aligned}
& (n+r-1)(2(r-1) + 2i-1) - \\
& 2(n+r-1)(j-1) - 2(m-1)(n+r-1) \geq 0, \\
& 2(n+r-1)(j-1) + 2m(n+r-1) - \\
& (n+r-1)(2(r-1) + 2i-1) \geq 0.
\end{aligned}$$

Therefore,

$$\begin{aligned}
& 2(r-1) + 2i-1 - 2(j-1) - 2(m-1) = \\
& 2r-2+2i-1-2j+2-2m+2 = \\
& 2r+2i-2j-2m+1 \geq 0, \\
& 2(j-1) + 2m - 2(r-1) - 2i+1 = \\
& 2j-2+2m-2r+2-2i+1 = \\
& 2j+2m-2r-2i+1 \geq 0.
\end{aligned}$$

Let  $i = m-r+j+a$ . In this case we have

$$2a+1 \geq 0,$$



$$-2a + 1 \geq 0.$$

So,  $a = 0$ . Therefore,  $i = m - r + j$ . Clearly,  $N_i$  belongs to the segment

$$[(2(n+r-1)(j-1) + 2(m-1)(n+r-1), 0);$$

$$(2(n+r-1)(j-1) + 2m(n+r-1), 0)]$$

if and only if

$$i = m - r + j.$$

Therefore, the point  $N_i$  is visible from the point  $S_j$  if and only if  $i \in X_j$ .  $\square$

#### IV. A LOGICAL MODEL FOR VL

Since the problem VL is **NP**-complete, there is no polynomial time algorithm for finding a solution for this problem. However, since the considered problem is of significant practical use, we need to find a fast algorithm for solving this problem.

Well known, many problems with practical applications belong to the class of computational complexity **NP**. Also all problems from the **NP** can be polynomially reduced to the problem of satisfiability of a Boolean function (SAT) (see e.g. [36]). This problem can be formulated as follows

**BOOLEAN SATISFIABILITY PROBLEM IN CONJUNCTIVE NORMAL FORM (SAT):**

INSTANCE: *A Boolean function*

$$g(x_1, x_2, \dots, x_n)$$

*in conjunctive normal form.*

QUESTION: *Is there an assignment of the set of variables such that*

$$g(x_1, x_2, \dots, x_n) = 1?$$

The satisfiability problem is a core problem in mathematical logic and computing theory. In practice, SAT is fundamental in solving many problems in automated reasoning, computer-aided design, computer-aided manufacturing, machine vision, database, robotics, integrated circuit design, computer architecture design, and computer network design. Traditional methods treat SAT as a discrete, constrained decision problem. Many optimization methods, parallel algorithms, and practical techniques have been developed for solving SAT. Recently, in the domain of development of fast algorithms to solving SAT was achieved a substantial progress (see, in particular, [37]). Most studies have focused on genetic algorithms and algorithms of local search. Note that significant attention is focused both on SAT and its optimization version MAXSAT (see e.g. [36]) which can be formulated as follows.

**MAXIMUM SATISFIABILITY PROBLEM IN CONJUNCTIVE NORMAL FORM (MAXSAT):**

INSTANCE: *A Boolean function*

$$g(x_1, x_2, \dots, x_n)$$

*in conjunctive normal form.*

QUESTION: *Is there an assignment of the set of variables such that in the function*

$$g(x_1, x_2, \dots, x_n)$$

*the maximum number of clauses is true?*

Recently, proposed several genetic algorithms [38] – [41]. Considered also hybrid algorithms based on combinations of genetic algorithms and local search algorithms [42]. Relatively high efficiency can be achieved for algorithms based solely on the local search. Of course, these algorithms run in exponential time in worst case. But they can relatively fast obtain a solution for many Boolean functions arising in practice. So, a reduction from hard problems to SAT and MAXSAT for its solving acquires a practical sense. For example, such approach was considered for hamiltonian path problem in [43], [44].

Consider a reduction the problem VL to the MAXSAT. For all  $i$ ,  $1 \leq i \leq n$ , consider a set

$$M_i = \{p \mid b_i \in F(a_p)\}.$$

It is easy to see that the system of sets  $M_i$ ,  $1 \leq i \leq n$ , can be constructed in polynomial time. Obviously, if for some  $i$  we have a following inequality  $|M_i| < d$ , then the solution for the problem VL is negative.

Thus, in further, without loss of generality, we assume that for all  $i$  it is true that  $|M_i| \geq d$ . Therefore, for all  $i$  we can consider the system

$$M_{i,1}, M_{i,2}, \dots$$

from

$$\frac{|M_i|(|M_i| - 1) \dots (|M_i| - d + 2)}{(d - 1)!}$$

pairwise different subsets of the set  $M_i$  each of which consists from  $|M_i| - d + 1$  elements.

Directly from the definition of  $M_{i,j}$  follows that if for some  $i$  and arbitrary  $j$  each of sets  $M_{i,j}$  contains at least one number of a point of a landmark location, then from the point  $b_i$  is visible at least  $d$  landmarks.

We interpret  $x_l = 1$  as presence of some landmark in the point with the number  $l$ . Hence, we obtain that

$$\bigwedge_{1 \leq j \leq \frac{|M_i|(|M_i|-1)\dots(|M_i|-d+2)}{(d-1)!}} \left( \bigvee_{l \in M_{i,j}} x_l \right)$$

is true if and only if from the point  $b_i$  is visible at least  $d$  landmarks.

Now we show that

$$\varphi = \left( \bigwedge_{\substack{1 \leq i \leq n, \\ 1 \leq r \leq m+1, \\ 1 \leq j \leq \frac{|M_i|(|M_i|-1)\dots(|M_i|-d+2)}{(d-1)!}}} \left( \bigvee_{l \in M_{i,j}} x_l \right) \vee s_r \right) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i \leq n, \\ 1 \leq r \leq m+1, \\ 1 \leq j \leq \frac{|M_i|(|M_i|-1)\dots(|M_i|-d+2)}{(d-1)!}}} \left( \bigvee_{l \in M_{i,j}} x_l \right) \vee \neg s_r \right) \wedge$$

$$\left( \bigwedge_{1 \leq t \leq m} \neg x_t \right)$$

provides a reduction from VL to MAXSAT.

Note that in  $\varphi$  the total number of clauses is equal to

$$2n(m+1) \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!} + m.$$

It is easy to see if there is a required placement of landmarks, then the Boolean function

$$\left( \bigwedge_{\substack{1 \leq i \leq n, \\ 1 \leq j \leq \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!}}} \bigvee_{l \in M_{i,j}} x_l \right) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i \leq n, \\ 1 \leq j \leq \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!}}} \bigvee_{l \in M_{i,j}} x_l \right)$$

is satisfiable. This obviously implies a satisfiability of the Boolean function

$$\left( \bigwedge_{\substack{1 \leq i \leq n, \\ 1 \leq r \leq m+1, \\ 1 \leq j \leq \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!}}} \left( \bigvee_{l \in M_{i,j}} x_l \right) \vee s_r \right) \wedge$$

$$\left( \bigwedge_{\substack{1 \leq i \leq n, \\ 1 \leq r \leq m+1, \\ 1 \leq j \leq \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!}}} \left( \bigvee_{l \in M_{i,j}} x_l \right) \vee \neg s_r \right).$$

If there is required placement of landmarks, then in the Boolean function  $\varphi$  at least

$$2n(m+1) \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!}$$

clauses is true.

Suppose, that there is no required placement of landmarks. Then there is  $i$  and  $j$  such that the Boolean function

$$\bigvee_{l \in M_{i,j}} x_l$$

is false. Therefore for all  $r$  either

$$\left( \bigvee_{l \in M_{i,j}} x_l \right) \vee s_r$$

or

$$\left( \bigvee_{l \in M_{i,j}} x_l \right) \vee \neg s_r$$

is false. Thus, in the Boolean function  $\varphi$  no more than

$$2n(m+1) \frac{|M_i|(|M_i|-1) \dots (|M_i|-d+2)}{(d-1)!} - 1$$

clauses is true. Therefore, we obtain a reduction from VL to MAXSAT.

## V. CONCLUSION AND EXPERIMENTAL RESULTS

In previous section we obtain an implicit reduction from VL to MAXSAT.

New algorithms allow ordinary desktop computers to solve boolean functions in conjunctive normal form, which has more than 10000 conjuncts (see e.g. [43]). There is a well known site on which posted solvers for SAT [45]. Currently on the site published 16 implementations of algorithms for solving SAT. They are divided into two main classes: stochastic local search algorithms and algorithms improved exhaustive search. All solvers allow the conventional format for recording DIMACS Boolean function in conjunctive normal form and solve the corresponding problem [46]. In addition to the solvers the site also represented a large set of test problems in the format of DIMACS. This set includes a randomly generated problems of satisfiability.

For the computational experiments we used heterogeneous cluster based on three clusters:

- The cluster of Ural State University (8 computational nodes, Linux, processor Intel Pentium IV 2.40GHz);
- The cluster umt of Institute of Mathematics and Mechanics, Ural Branch of the Russian Academy of Sciences (256 computational nodes, Linux, processor Xeon 3.00GHz) [47] (see also [48]);
- The cluster um64 of Institute of Mathematics and Mechanics, Ural Branch of the Russian Academy of Sciences (124 computational nodes, Linux, dual-core processor AMD Opteron 2.6GHz bi-processors) [47] (see also [48]).

In our experiments we used own genetic algorithm MSAT and two standard solvers [45] (fgrasp and posit). Computational experiments were carried out on standard tests [45] and tests, obtained by special generators creating a natural data for the problem VL and for a number of others robotics problems.

The total was carried out 14 runs of cluster in the format of 100 nodes on 20 hours. For summarizing final statistics were selected 200 tests. Chosen tests was counterated by three solvers in 5 modes with constraints on the limit of time

- no restriction;
- 10 seconds;
- 100 seconds;
- 500 seconds;
- 1000 seconds.

We found that all three solvers have about the same performance. The best average velocity is showed by fgrasp. A slightly lower average velocity is showed by posit. Genetic algorithm showed a worst average velocity. From other hand we found that algorithms of local search are more resource demanding then the genetic algorithm. In particular, on some tests fgrasp could not finish the execution.

From our experiments we obtain an important property of genetic algorithm. Algorithms of local search showing relatively smooth results on several tests but genetic algorithm has a significant difference (from 26 seconds to 12 hours) in time during the test execution. The total trend can be described as follows. In many cases run time ranges from few seconds

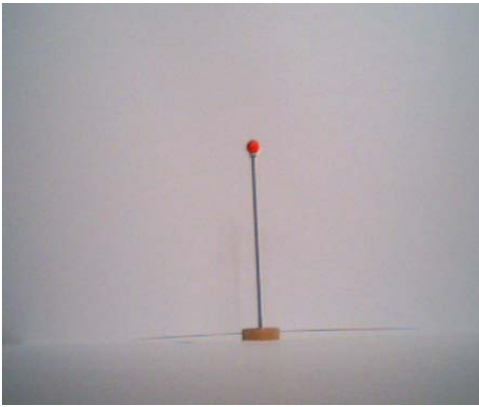


Fig. 1. An artificial landmark.



Fig. 2. Semi-artificial landmarks.

to few minutes. However, on a small set of Boolean functions a run time of genetic algorithm increased to 10-12 hours.

Our experiments have shown that our approach can be used for design of an efficient solver for the problem of placement of visual landmarks. Further advantages we can get by improving our genetic algorithm.

The first author developed a software package for processing video data to compute three-dimensional coordinates of objects [49]. This software package is designed for visual navigation on landmarks. To date, an intelligent system allowing the use of artificial (specially designed) and semi-artificial (located in a special way but not specifically designed) landmarks of various types implemented (e.g. Fig. 1 and 2).

Module of visual navigation based on landmarks used as part of the onboard control system of various modifications of robots Kuzma-I (e.g. Fig. 3 and 4) and Kuzma-II (e.g. Fig. 5 and 6). The onboard control system with module of visual navigation based on landmarks of a modifications of Kuzma-II (Fig. 5) was demonstrated at the International Exhibition INNOPROM – 2010 (15.07.2010 – 17.07.2010).

For our experiments on intelligent control systems, we use



Fig. 3. Robot Kuzma-I. Design of this robot based on the well-known RC cars. From RC-CAR AT-10ES Thunder Tiger [50] we use only the four wheel chassis, the high torque DC-MOTOR and a steering servo. The DC-MOTOR drives the chassis and a steering servo controls the direction. The electronic system based on SSC-32 microcontroller. Onboard computer based on a motherboard with x86 compatible processor AMD Geode LX600 for embedded systems. The robot is equipped with USB web camera Live! Cam Video IM Pro (VF0410) [52].

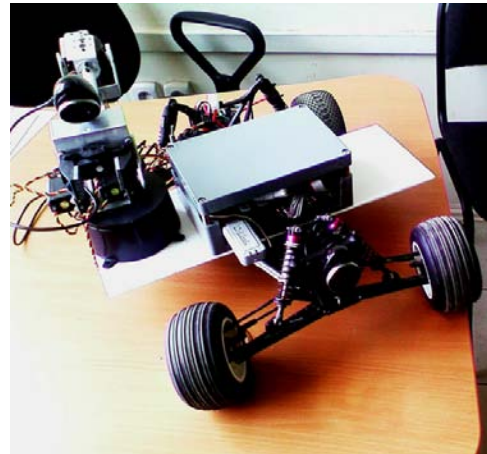


Fig. 4. Another modification of Kuzma-I. The robot is equipped with modified Lynxmotion [51] robotic arm and 2 x USB web camera Live! Cam Video IM Pro (VF0410).

a testbed composed of these mobile robots and a stationary monitoring system. In particular, we study different algorithms of visual navigation based on landmarks. In general, good results of robotic experiments do not guarantee high efficiency of algorithms. Perhaps experiments are conducted in too simple environments. Our theoretical results help us to select appropriate methods as well as testbeds to demonstrate them. Some of our robots are able to add their own landmarks (e.g. Fig. 4 and 5). They use a wireless connection to a supercomputer to run the solver for VL to plan their actions.

#### ACKNOWLEDGMENT

The work was partially supported by Grant of President of the Russian Federation MD-1687.2008.9 and Analytical Departmental Program "Developing the scientific potential of high school" 2.1.1/1775.



Fig. 5. Robot Kuzma-II. Design of this robot based on the well-known Johnny 5 Robot [53]. By utilizing heavy duty polypropylene and rubber tracks with durable ABS molded sprockets the robot has excellent traction. It includes two 12vdc 50:1 gear head motors and the Sabertooth 2 x 5 R/C motor controller. The electronic system based on SSC-32 microcontroller. Onboard computer of this robot is Asus Eee PC 1000HE. The robot is equipped with modified Lynxmotion robotic arm with wrist rotate and USB web camera Live! Cam Video IM Pro (VF0410).

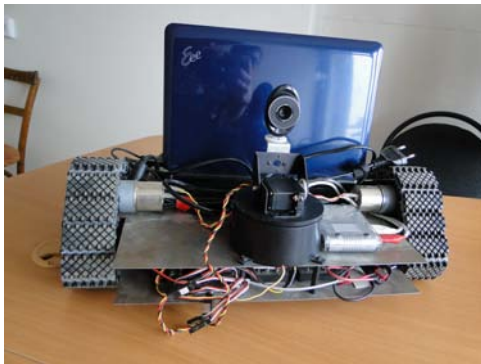


Fig. 6. Another modification of Kuzma-II. The robot is equipped with a 2 DOF robotic camera (USB web camera Live! Cam Video IM Pro (VF0410)).

## REFERENCES

- [1] Rus D., Vona M. *Crystalline Robots: Self-reconfiguration with Unit-compressible Modules*. Autonomous Robots. 2001. Vol. 10(1). P. 107–124.
- [2] Ünsal C., Khosla P.K. *A Multi-layered Planner for Self-Reconfiguration of a Uniform Group of I-Cube Modules*. IEEE International Conference on Intelligent Robots and Systems. 2001. Vol. 1. P. 598–605.
- [3] Christensen D., Ostergaard E., Lund H.H. *Metamodule control for the atron self-reconfigurable robotic system*. Proceedings of the The 8th Conference on Intelligent Autonomous Systems. 2004. P. 685–692.
- [4] Dewey D., Srinivasa S.S., Ashley-Rollman M.P., De Rosa M., Pillai P., Mowry T.C., Campbell J.D., Goldstein S.C. *Generalizing Metamodules to Simplify Planning in Modular Robotic Systems*. Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems. 2008. P. 1338–1345.
- [5] Ashley-Rollman M., Goldstein S., Lee P., Mowry T., Pillai P. *Meld: A declarative approach to programming ensembles*. Proceedings of the IEEE International Conference on Robots and Systems. 2007. P. 2794–2800.
- [6] Charron-Bost B., Delporte-Gallet C., Fauconnier H. *Local and temporal predicates in distributed systems*. ACM Transactions on Programming Languages and Systems. 1995. Vol. 17(1). P. 157–179.
- [7] Lowe D. *Object Recognition from Local Scale-Invariant Features*, ICCV. 1999. P. 1150–1157.
- [8] Dudek G., Jugessur D. *Robust Place Recognition using Local Appearance based Methods*. Proceedings of 2000 IEEE International Conference on Robotics and Automation. 2000. P. 1030–1035.
- [9] Carneiro G., Jepson A.D. *Multi-scale Phase-based Local Features*. CVPR. 2003. P. 736–743.
- [10] Yang G., Hou Z.-G., Liang Z. *Distributed visual navigation based on neural Q-learning for a mobile robot*. International Journal of Vehicle Autonomous Systems. 2006. Vol. 4(2-4). P. 225–238.
- [11] *Visual navigation system for a mobile robot having capabilities of regenerating of hidden images*. United States Patent 4887223.
- [12] Graf B. *Reactive navigation of an intelligent robotic walking aid*. Proceedings of the 2001 IEEE International Workshop on Robot Human Interaction. 2001. P. 353–358.
- [13] Pal P.K., Kar A. *Sonar-Based Mobile Robot Navigation Through Supervised Learning on a Neural Net*. Autonomous Robots. 1996. Vol. 3. P. 355–374.
- [14] Saint-Bauzel L., Pasqui V., Monteil I. *A reactive robotized interface for lower limb rehabilitation: clinical results*. IEEE Transactions on Robotics. 2009. Vol. 25(3). P. 583–592.
- [15] Angeli A., Filliat D., Doncieux S., Meyer J.-A. *Visual topological SLAM and global localization*. Proceedings of the International Conference on Robotics and Automation. 2009. P. 1–6.
- [16] Angeli A., Filliat D., Doncieux S., Meyer J.-A. *Incremental vision-based topological SLAM*. Proceedings of the 2008 IEEE International Conference on Intelligent Robots and Systems. 2008. P. 1–6.
- [17] Filliat D. *Interactive learning of visual topological navigation*. Proceedings of the 2008 IEEE International Conference on Intelligent Robots and Systems. 2008. P. 1–7.
- [18] Goedemé T., Nuttin M., Tuytelaars T., Van Gool L. *Omnidirectional Vision Based Topological Navigation*. International Journal of Computer Vision. 2007. Vol. 74(3). P. 219–236.
- [19] Vale A., Isabel Ribeiro M. *Environment Mapping as a Topological Representation*. Proceedings of the 11th International Conference on Advanced Robotics. 2003. P. 1–6.
- [20] Kazumi O., Hidenori T., Takanori E., Takeshi T., Shigenori O. *Navigation using Artificial Landmark without using Coordinate System*. Nippon Robotto Gakkai Gakujutsu Koenkai Yokoshu. 2003. Vol. 21. P. 3J11.
- [21] Samuelsson M. *Artificial landmark navigation of an autonomous robot*. Master Thesis. Örebro University, Department of technology, SE-70182. Örebro, Sweden, 2005.
- [22] Hellmann I., Siemiatkowska B. *Artificial landmark navigation system*. International symposium on intelligent robotic systems N 9. 2001. P. 219–228.
- [23] Åstrand B., Baerveldt A.-J. *An Agricultural Mobile Robot with Vision-Based Perception for Mechanical Weed Control*. Autonomous Robots. 2002. Vol. 13. P. 21–35.
- [24] Hagrais H., Callaghan V., Colley M. *Prototyping design and learning in outdoor mobile robots operating in unstructured outdoor environments*. IEEE International Robotics and Automation Magazine. 2001. Vol. 8(3). P. 53–69.
- [25] Hagrais H., Colley M., Callaghan V. *Online Learning and Adaptation of Autonomous Mobile Robots for Sustainable Agriculture*. Autonomous Robots. 2002. Vol. 13. P. 37–52.
- [26] Jung H.C. *Visual Navigation for a Mobile Robot Using Landmarks*. Advanced Robotics. 1994. Vol. 9(4). P. 429–442.
- [27] Gilg A., Schmidt G. *Landmark-oriented visual navigation of a mobile robot*. IEEE International Symposium on Industrial Electronics. 1993. P. 257–262.
- [28] Zhu Z., Oskiper T., Samarasekera S., Kumar R. *Precise visual navigation using multi-stereo vision and landmark matching*. Proceedings of the SPIE. 2007. Vol. 6561. P. 656108.
- [29] Murrieta-Cid R., Parra C., Devy M. *Visual navigation in natural environments: from range and color data to a landmark-based model*. Autonomous Robots. 2002. V. 13(2). P. 143–168.
- [30] Sala P., Sim R., Shokoufandeh A., Dickinson S. *Landmark Selection for Vision-Based Navigation*. IEEE Trans. on Robotics. 2006. V. 22(2). P. 334–349.
- [31] Basri R., Rivlin E. *Localization and Homing Using Combinations of Model Views*. AI. 1995. Vol. 78(1-2). P. 327–354.
- [32] Wilkes D., Dickinson S., Rivlin E., Basri R. *Navigation Based on a Network of 2D Images*. ICPR-A. 1994. P. 373–378.

- [33] Popov V. *The approximate period problem for DNA alphabet*. Theoretical Computer Science. 2003. Vol. 304. P. 443–447.
- [34] Popov V. *The Approximate Period Problem*. IAENG International Journal of Computer Science. 2009. Vol. 36(4). P. 268–274.
- [35] Garey M.R., Johnson D.S. *Computers and Intractability. A Guide to the Theory of NP-completeness*. W. H. Freeman, California, 1979.
- [36] Papadimitriou C.H. *Computational Complexity*. Addison-Wesley Publishing Company, Reading/Menlo Park, NY, 1994.
- [37] Gu J., Purdom P., Franco J., Wah B. *Algorithms for the Satisfiability (SAT) Problem: A Survey*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1996. P. 19–152.
- [38] Fleurent J. *Genetic algorithms and hybrids for graph coloring*. Annals of Operations Research. 1996. Vol. 63. P. 437–461.
- [39] Hao J., Dorne R. *A new population-based method for satisfiability problems*. Proceedings of 11th European Conference on Artificial Intelligence. 1994. P. 135–139.
- [40] Jong K., Spears W. *Using genetic algorithms to solve np-complete problems*. Proceedings of the International Conference on Genetic Algorithms. 1989. P. 124–132.
- [41] Voorn R., Dastani M., Marchiori E. *Finding simplest pattern structures using genetic programming*. Proceedings of the Genetic and Evolutionary Computation Conference. 2001. P. 3–10.
- [42] Hao J., Lardeux F., Saubion F. *A hybrid genetic algorithm for the satisfiability problem*. Proceedings of the 1st International Workshop on Heuristics. 2002. P. 102–109.
- [43] Iwama K., Miyazaki S. *SAR-variable complexity of hard combinatorial problems*. IFIP Trans. A Comput. Sci. Tech. 1994. Vol. I. P. 253–258.
- [44] Plotnikov A.D. *A Logical Model of HCP*. International Journal of Mathematics and Mathematical Sciences. 2001. Vol. 26(11).
- [45] <http://people.cs.ubc.ca/~hoos/SATLIB/index-ubc.html>
- [46] <http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps>
- [47] [http://parallel.imm.uran.ru/mvc\\_now/hardware/supercomp.htm](http://parallel.imm.uran.ru/mvc_now/hardware/supercomp.htm)
- [48] <http://parallel.uran.ru/node/6>
- [49] Gorbenko A. *Software for processing video data to compute three-dimensional coordinates of objects*. Bachelor Thesis, Department of Mathematics and Mechanics, Ural State University, Ekaterinburg, 2009. (in russian)
- [50] <http://www.tiger.com.tw/>
- [51] <http://www.lynxmotion.com/>
- [52] [http://support.creative.com/Products/ProductDetails.aspx?%20catID=218&CatName=Web+Cameras&subCatID=%20846&subCatName=Live!+Cam+Series&prodID=16904&prodName=Live!+Cam+Video+IM+Pro+\(VF0410\)&bTopTwenty=1&VARSET=prodfaq:PRODFAQ\\_16904,VARSET=CategoryID:218](http://support.creative.com/Products/ProductDetails.aspx?%20catID=218&CatName=Web+Cameras&subCatID=%20846&subCatName=Live!+Cam+Series&prodID=16904&prodName=Live!+Cam+Video+IM+Pro+(VF0410)&bTopTwenty=1&VARSET=prodfaq:PRODFAQ_16904,VARSET=CategoryID:218)
- [53] <http://www.lynxmotion.com/c-103-johnny-5.aspx>

# Hand Recognition in Live-Streaming Video

Mikhail Belov

Department of business-informatics  
Higher School of Economics  
Moscow, Russian Federation  
mpbelov@gmail.com

**Abstract**— The article describes the algorithmic component of the pattern recognition method for extracting hand patterns from a video stream. Methods removing excess information from frames, localizing fragments with a hand and extracting hand contours to classify them are described.

**Keywords**-pattern recognition; Hu invariants; Canny detector; video stream processing

## I. INTRODUCTION

One can input data into a computer in a form of graphical information. There are methods for processing the graphical information and for treating it not only as a set of dots with color codes, but also as a container for another data. This fact gives an opportunity to extend the number of human computer interaction (HCI) ways. Such systems are described in [5] and [6].

It is planned to develop a system prototype for direct and online controlling the graphical objects displayed on a screen. Currently this idea has been implemented in two types of systems: sensor screens and “smart boards”. In the first case, a transparent sensor pad is placed over the screen, which catches the user’s touches and translates them into control signals to the processor. Due to an existing technology such screens are expensive and produced mostly in small and medium formats. In case of the “smart boards” the projector’s light is not focused on a usual board, but aimed to a special sensor surface. Unlike sensor screens, there are large “smart boards” because of projector, but this approach remains rather expensive and not suitable as a mass solution. Instead, building such a HCI system based on a video camera and a projector will reduce the dependency of the cost from the display size.

The article describes the algorithmic component of the pattern recognition method for extracting hand patterns from a video stream.

## II. PREPARING A FRAME

To remove excess information from a frame one can use the Histogram Backprojection method. In this case a hand is being searched by its color characteristics. The method can be applied to search for pixels satisfying the histogram, or to search a pattern (of a hand image) by shifting the pattern w.r.t. the initial image. A frame fragment containing the hand image should be used as a template histogram.

In Histogram Backprojection the model (target) and the image are represented by their multidimensional color histograms  $M$  and  $I$  as in Histogram Intersection. A ratio histogram  $R$ , defined as  $R_i = \min\left(\frac{M_i}{I_i}, 1\right)$ , is computed from the model and image histograms. It is this histogram  $R$  that is backprojected onto the image, that is, the image values are replaced by the values of  $R$  that they index. The backprojected image is then convolved by a mask, which for compact objects of unknown orientation could be a circle with the same area as the expected area subtended by the object. The peak in the convolved image is the expected location of the target, provided the target appears in the image [1].

## III. LOCALIZING A FRAME FRAGMENT WITH A HAND

We can locate a fragment with hand by calculating the difference image characteristics [4]. We have to use an image template, which contains a hand picture.

One of possible methods is a square difference matching method. Perfect match leads to 0 result. Large result means bad match:

$$R_{sqdiff}(x, y) = \sum_{x', y'} [T(x', y') - I(x + x', y + y')]^2$$

Correlation matching methods multiplicatively match the template against the image so a perfect match will be large and bad matches will be small or 0.

$$R_{ccorr}(x, y) = \sum_{x', y'} [T(x', y') \cdot I(x + x', y + y')]^2$$

Correlation coefficient matching methods match a template relative to its mean against the image relative to its mean, so a perfect match will be 1 and a perfect mismatch will be  $-1$ ; a value of 0 simply means that there is no correlation (random alignments).

$$R_{ccoeff}(x, y) = \sum_{x', y'} [T'(x', y') \cdot I'(x + x', y + y')]^2$$

$$T'(x', y') = T(x', y') - \frac{1}{(w \cdot h) \sum_{x'', y''} T(x'', y'')} \\ I(x + x, y + y) = \\ = I(x + x, y + y) - \frac{1}{(w \cdot h) \sum_{x, y} I(x + x, y + y)}$$

These factors may be normalized [4]. The normalized methods are useful because they can help reduce the effects of lighting differences between the template and the image. In each case, the normalization coefficient is the same:

$$Z(x, y) = \sqrt{\sum_{x', y'} T(x', y')^2 \cdot \sum_{x', y'} I(x + x', y + y')^2}$$

#### IV. EXTRACTING HAND CONTOURS

Then we extract contours from the image using the Canny detector [2]. The Canny algorithm runs in 5 separate steps.

##### A. Smoothing: Blurring of the image to remove noise

It is inevitable that all images taken from a camera will contain some amount of noise. To prevent that noise is mistaken for edges, noise must be reduced. Therefore the image is first smoothed by applying a Gaussian filter.

##### B. Finding gradients: The edges should be marked where the gradients of the image has large magnitudes

Gradients at each pixel in the smoothed image are determined by applying what is known as the Sobel-operator.

##### C. Non-maximum suppression: Only local maxima should be marked as edges

The purpose of this step is to convert the “blurred” edges in the image of the gradient magnitudes to “sharp” edges. Basically this is done by preserving all local maxima in the gradient image, and deleting everything else. The algorithm is for each pixel in the gradient image:

- 1) Round the gradient direction  $\theta$  to nearest  $45^\circ$ , corresponding to the use of an 8-connected neighbourhood;
- 2) Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient direction. I.e. if the gradient direction is north ( $\theta = 90^\circ$ ), compare with the pixels to the north and south;
- 3) If the edge strength of the current pixel is largest; preserve the value of the edge strength. If not, suppress (i.e. remove) the value.

##### D. Double thresholding: Potential edges are determined by thresholding

Edge pixels stronger than the high threshold are marked as strong; edge pixels weaker than the low threshold are suppressed and edge pixels between the two thresholds are marked as weak.

##### E. Edge tracking by hysteresis: Final edges are determined by suppressing all edges that are not connected to a very certain (strong) edge

Strong edges are interpreted as “certain edges”, and can immediately be included in the final edge image. Weak edges are included if and only if they are connected to strong edges. The logic is of course that noise and other small variations are unlikely to result in a strong edge (with proper adjustment of the threshold levels). Thus strong edges will (almost) only be due to true edges in the original image. The weak edges can either be due to true edges or noise/color variations.

#### V. CLASSIFYING THE FOUND CONTOUR

We classify found contours after the extraction. Hu invariant moments of contours are used for this. Moment is a gross characteristic of the contour computed by integrating over all of the pixels of the contour [3]. The  $(p, q)$  moment of a contour is defined as:

$$m_{p,q} = \sum_{i=1}^n I(x, y) x^p y^q$$

Here  $p$  is the x-order and  $q$  is the y-order, whereby order means the power to which the corresponding component is taken in the sum just displayed. The summation is over all of the pixels of the contour boundary (denoted by  $n$  in the equation).

The moment computation just described gives some rudimentary characteristics of a contour that can be used to compare two contours. However, the moments resulting from that computation are not the best parameters for such comparisons in most practical cases. In particular, one would oft en like to use normalized moments (so that objects of the same shape but dissimilar sizes give similar values). Similarly, the simple moments of the previous section depend on the coordinate system chosen, which means that objects are not matched correctly if they are rotated.

A central moment is basically the same as the moments just described except that the values of  $x$  and  $y$  used in the formulas are displaced by the mean values:

$$\mu_{p,q} = \sum_{i=1}^n I(x, y) (x_{avg})^p (y - y_{avg})^q$$

where  $x_{avg} = \frac{m_{10}}{m_{00}}$  and  $y_{avg} = \frac{m_{01}}{m_{00}}$ .

The normalized moments are the same as the central moments except that they are all divided by an appropriate power of  $m_{00}$ :

$$\eta_{p,q} = \frac{\mu_{p,q}}{m_{00}^{(p+q)/2+1}}$$

The Hu invariant moments are linear combinations of the central moments.

The following factors are used to detect similarity between two contours [4]:

$$I_1(A, B) = \sum_{i=1}^7 \left| \frac{1}{m_i^A} - \frac{1}{m_i^B} \right|$$

$$I_2(A, B) = \sum_{i=1}^7 |m_i^A - m_i^B|$$

$$I_3(A, B) = \sum_{i=1}^7 \left| \frac{m_i^A - m_i^B}{m_i^A} \right|$$

Here  $m_i^A$  and  $m_i^B$  are defined as:



$$m_i^A = \text{sign}(h_i^A) \cdot \log|h_i^A|,$$

$$m_i^B = \text{sign}(h_i^B) \cdot \log|h_i^B|,$$

where  $h_i^A$  and  $h_i^B$  are the Hu moments of A and B, respectively.

After classification system performs the action associated with a certain gesture.

## VI. FUTURE WORK

The described image processing methods may be used with various parameters. It is planned to implement the investigated algorithm and to choose the best methods among the available alternatives at the next step of the research. It is also needed to assess the capabilities of usage Hu moments for hand contour comparison and similarity detection. As further research of the described method it is required to compare the contour

similarity coefficient metrics listed above based on Hu moments by quality of classification result.

## REFERENCES

- [1] M. Swain, D. Ballard, "Color Indexing". International Journal of Computer Vision, 7:1, Kluwer Academic Publishers, Manufactured in The Netherlands, 1991, pp. 11-32.
- [2] J. Canny. "A computational approach to edge detection. Pattern Analysis and Machine Intelligence", IEEE Transactions on, PAMI-8(6), Nov. 1986, pp. 679-698.
- [3] M. Hu, "Visual pattern recognition by moment invariants" IRE Transactions on Information Theory 8, 1962, pp. 179-187
- [4] G. Bradski, A. Kaehler. "Learning OpenCV". O'Reilly Media. 2008. Pages: 576.
- [5] P. Garg, N. Aggarwal, S. Sofat. "Vision Based Hand Gesture Recognition", World Academy of Science, Engineering and Technology - 49, 2009.
- [6] C. Keskin, A. Erkan, L. Akarun, "Real Time Hand Tracking and 3D Gesture Recognition for Interactive Interfaces Using Hmm", ICANN/ICONIPP, 2003.



# 3<sup>D</sup>-Illusion Constructor

Maksim Rovkin, Evgenij Yel'chugin, Maria Filatova

dept. of Mathematics and Mechanics

Urals State University

Ekaterinburg, Russian Federation

e-mail : Maria.Filatova@usu.ru

**Abstract**—Madonnari, one of the kinds of street art, is very popular in the world today. Drawings are made on a pavement and are deliberately distorted in such a way that an illusion of a three-dimensional object appears when the drawing is looked at from a certain point. It is remarkable that many painters use methods developed in the sixteenth century to make such drawings. Some graphical packages, for example Photoshop, allow to distort images in such a way that they can be seen correctly from a different point. However, such packages cannot be used to construct illusions at the junction of two plains, and a viewing point cannot be prescribed. We present a program which makes it possible to construct 3D illusions at the junction of two or three plains and to select a point, viewing from which produces a spatial effect.

*Image processing; computer graphics; graphic design; projective geometry; estimation*

## I. INTRODUCTION

Anamorphosis is an art of construction of deliberately distorted images which, when looked at from a certain point, regain their undistorted view. The art of anamorphosis was invented in China and brought to Italy in the sixteenth century. Probably one of the most spectacular and impressing examples of anamorphosis is Madonnari – one of the kinds of street art. Drawing are made on a pavement (we will call such drawings 3D-illusions) and are deliberately distorted in such a way that an illusion of a three-dimensional object appears when the drawing is looked at a certain angle. The central object of art in Italy in the sixteenth century was Madonna, which is the reason for the name “Madonnari”.

This kind of art is very popular in our time; moreover, many businesses use 3D-illusions for advertising. Examples of such illusions is shown in Figure 1. Recently an experiment was started in Canada, in which the speed of vehicles is controlled by an optical illusion made on the road. (This experiment is conducted by the Fund of Traffic Security of British Columbia.)

It is surprising that many painters that draw illusions on a pavement use the same methods that were used 400 years ago by their predecessors. Some graphical packages, such as Adobe Photoshop, have a Perspective tool, which allows to construct 3D-illusions in a plane. However, this tool is designed to remove perspective distortions rather than to construct illusions that appear as a result of perspective distortions. For this reason it is not possible to prescribe a point, viewing from which produces an illusion. It is not possible either to construct an

illusion from a number of images (or regions of images) that lie in different planes. Thus we are not aware of existence of a graphical package that is designed to construct 3D-illusions in a plane or a number of planes (although we investigate this question specifically). Despite simplicity of realization, we found the idea of creating such a graphical tool very interesting.

We present a program (a 3D-illusion constructor) written in C#, which allows to produce 3D-illusions in a plane.



Figure 1. Examples of such illusions advertising

## II. MATHEMATICAL MODEL

Suppose that, in three planes, we want to obtain an illusion of the image in Picture. Consider a Cartesian system in the space (see Figure 2). Thus the problem is to construct a projection of the Picture plane to the coordinate planes.

Let  $(u_1, u_2)$  coordinates of a pixel in Picture plane,  $(v_1; v_2)$  coordinates of a pixel in one of coordinate planes.

Our task is to obtain a projective transformation between two planes. Such transformation is a rational function of the form

$$u_i = \frac{k_i^1 v_1 + k_i^2 v_2 + d_i}{k_i^3 v_1 + k_i^4 v_2 + 1}, \quad (1)$$
$$i = 1, 2$$

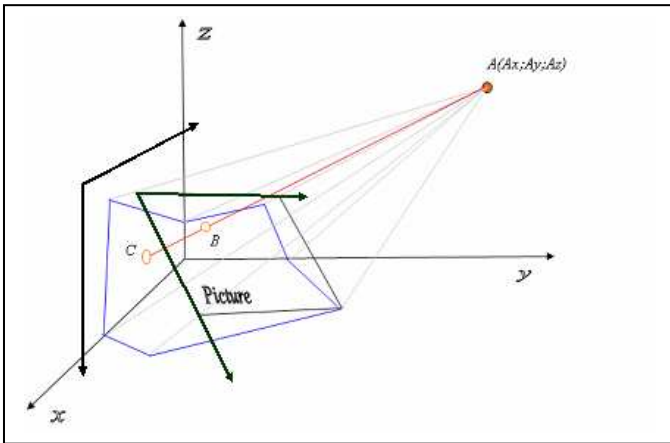


Figure 2. a projection of the Picture plane to the coordinate planes

To find ten coefficients  $k_i^j$ ,  $i=1,2$   $j=1..5$  we have to know a set of five points  $(u_1, u_2)$  and a corresponding set of points  $(v_1, v_2)$ . These sets can be found using geometric view. If we substitute these points in (1) we obtain a system of linear equations. The variables of this system are coefficients  $k_i^j$ ,  $i=1,2$   $j=1..5$  se in an equation.



Figure 3. Resulting image

### III. PROGRAM DESCRIPTION

The program works with raster images. The user can give the coordinates of the point A and the size of the file, where the illusion will be constructed. After that the images can selected, which can placed in a 3D space, dimensions can modified, and

they can projected to a plane (or planes), where the illusion is constructed. The output of the program is an image file. After the output image is printed and viewed from a certain point, an illusion appears.

The results obtained with the program are shown in Figures 3 and 4. If the image of Figure 3 is printed, cut angle and viewed at a certain point, this produces an illusion shown in Figure 4. We also note that this program allows to produce cards shown in Figure 5.



Figure 4. Illusion image

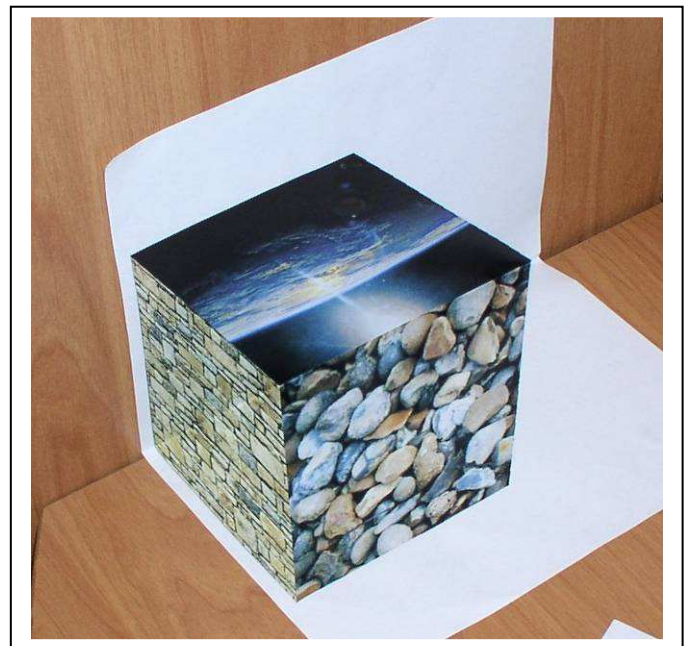


Figure 5. Illusion card

[1] Richard Hartley, and Andrew Zisserman, "Multiple View Geometry in Computer Vision," Cambridge University Press 2000, pp. 607