

# **SYRCoSE 2016**

Editors:

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

Preliminary Proceedings of the 10<sup>th</sup> Spring/Summer Young Researchers'  
Colloquium on Software Engineering

Krasnovidovo, May 30-June 1, 2016

**Preliminary Proceedings of the 10<sup>th</sup> Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2016), May 30-June 1, 2016 – Krasnovidovo, Mozhaysky District, Moscow Oblast, Russia.**

The issue contains papers accepted for presentation at the 10<sup>th</sup> Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2016) held in Krasnovidovo, Mozhaysky District, Moscow Oblast, Russia on May 30-June 1, 2016. The paper selection was based on originality and contributions to the field. Each paper was peer-reviewed by at least three referees.

The colloquium's topics include programming languages, software development tools, embedded and cyber-physical systems, software and hardware verification, formal methods, information security, and others.

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

The event is sponsored by Russian Foundation for Basic Research (Project №16-07-20256).

# Contents

Foreword .....	6
Committees .....	7
Referees .....	8
Language Support for Generic Programming in Object-Oriented Languages: Design Challenges <i>J. Belyakova</i> .....	9
Refinement Types in Jolie <i>A. Tchitchigin, L. Safina, M. Elwakil, M. Mazzara, F. Montesi, V. Rivera</i> .....	20
Visual Dataflow Language for Educational Robots Programming <i>G. Zimin, D. Mordvinov</i> .....	24
Programming Languages Segmentation via the Data Mining Software “ShaMaN” <i>T. Afanasieva, S. Makarova, D. Shalaev, A. Efremov</i> .....	32
Context-Based Model for Concern Markup of a Source Code <i>M. Malevannyy, S. Mikhalkovich</i> .....	39
Metric-Based Approach to Anti-Pattern Detection in Service Oriented Software Systems <i>A. Yugov</i> .....	45
Technology for Application Family Creation Based on Domain Analysis <i>A. Gudoshnikova, Yu. Litvinov</i> .....	52
Language for Describing Templates for Test Program Generation for Microprocessors <i>A. Tatarnikov</i> .....	59
Specification-Based Test Program Generation for MIPS64 Memory Management Units <i>A. Kamkin, A. Kotsynyak</i> .....	68
Approaches to Stand-alone Verification of Multicore Microprocessor Caches <i>M. Petrochenkov, I. Stotland, R. Mushtakov</i> .....	73
Checking Parameterized PROMELA Models of Cache Coherence Protocols <i>V. Burenkov, A. Kamkin</i> .....	77
A Model Checking-Based Method of Functional Test Generation for HDL Descriptions <i>M. Lebedev, S. Smolov</i> .....	84
Deriving Adaptive Checking Sequence for Nondeterministic Finite State Machines <i>A. Ermakov, N. Yevtushenko</i> .....	90
Conversion of Abstract Behavioral Scenarios into Scenarios Applicable for Testing <i>P. Drobintsev, V. Kotlyarov, I. Nikiforov, N. Voinov, I. Selin</i> .....	96
Automation of Failure Mode, Effects and Criticality Analysis <i>P. Privalov</i> .....	104

Parallel Processing and Visualization for Results of Molecular Simulations Problems <i>D. Puzyrkov, V. Podryga, S. Polyakov</i>	108
Memristor-based Hardware Neural Networks Modelling Review and Framework Concept <i>D. Kozhevnikov, N. Krasilich</i>	118
A Method of Converting an Expert Opinion to Z-number <i>E. Glukhoded, S. Smetanin</i>	124
Development and Research of Models of Self-Organization of Data Placement in Software-Defined Infrastructures of Virtual Data Center <i>I. Bolodurina, D. Parfenov</i>	129
Automated Text Document Compliance Assessment System <i>M. Zhigalova, A. Sukhov</i>	135
Complete Contracts through Specification Drivers <i>A. Naumchev, B. Meyer</i>	141
Usability of AutoProof: a Case Study of Software Verification <i>M. Khazeev, V. Rivera, M. Mazzara, A. Tchitchigin</i>	149
Certified Grammar Transformation to Chomsky Normal Form in $F^*$ <i>M. Polubelova, S. Bozhko, S. Grigorev</i>	155
Performance Testing of Automated Theorem Provers Based on Sudoku Puzzle <i>M. Sabyanin, D. Senotov, G. Skvortsov, R. Yavorsky</i>	160
Translation of Nested Petri Nets into Petri Nets for Unfoldings Verification <i>V. Ermakova, I. Lomazova</i>	164
Automatic Code Generation from Nested Petri Nets to Event-based Systems on the Telegram Platform <i>D. Samokhvalov, L. Dworzanski</i>	173
Mining Hierarchical UML Sequence Diagrams from Event Logs of SOA Systems while Balancing between Abstracted and Detailed Models <i>K. Davydova, S. Shershakov</i>	181
Applying MapReduce to Conformance Checking <i>I. Shugurov, A. Mitsyuk</i>	189
Modelling the People Recognition Pipeline in Access Control Systems <i>F. Gossen, T. Margaria, T. Göke</i>	198
System for Deep Web Users Deanonimization <i>A. Lazarenko, S. Avdoshin</i>	206
Model of Security for Object-Oriented and Object-Attributed Applications <i>P. Oleynik, S. Salibekyan</i>	211
Dynamic Key Generation According to the Starting Time <i>A. Kiryantsev, I. Stefanova</i>	217
Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS <i>C.T. Hansen, P.G. Larsen</i>	223



A Static Approach to Estimation of Execution Time of Components in AADL Models <i>A. Troitskiy, D. Buzdalov</i> .....	229
Practical Experience of Software and System Engineering Approaches in Requirements Management for Software Development in Aviation Industry <i>I. Koverninskiy, A. Kan, V. Volkov, Yu. Popov, N. Gorelits</i> .....	236
Design and Architecture of Real-time Operating System <i>K. Mallachiev, N. Pakulin, A. Khoroshilov</i> .....	239
Developing a Debugger for Real-Time Operating System <i>A. Emelenko, K. Mallachiev, N. Pakulin</i> .....	245
Building and Testing an Embedded Operating System <i>A. Ovcharov, N. Pakulin</i> .....	250

# Foreword

Dear participants, it is our pleasure to meet you at the 10<sup>th</sup> Anniversary Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE). This year's colloquium is hosted by Moscow State University (MSU), one of the oldest (established in 1755 by M.V Lomonosov), biggest and most famous Russian universities. Graduates of the university greatly contributed to theoretical computer science, system programming, and software engineering. Among them are Professors A.P. Ershov (1931 – 1988), L.N. Korolev (1926 – 2016), V.V. Lipaev (1928 – 2015), A.A. Lyapunov (1911 – 1973), E.Z. Lyubimskiy (1931 – 2008), V.S. Shtarkman (1931 – 2005), M.R. Shura-Bura (1918 – 2008), V.F. Turchin (1931 – 2010), E.A. Zhogolev (1930 – 2003), and many others. SYRCoSE 2016 is organized by Institute for System Programming of the Russian Academy of Sciences (ISP RAS) jointly with Moscow and Saint-Petersburg Universities.

SYRCoSE 2016's Program Committee (consisting of more than 50 members from more than 25 organizations) has selected 38 papers. Each submitted paper has been reviewed independently by three referees. The authors and speakers represent well-known universities, research institutes and companies including Aarhus University, Aston, Cairo University, Higher School of Economics, Innopolis University, Institute for System Programming of the Russian Academy of Sciences, Kazan Federal University, Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences, Lero – The Irish Software Research Centre, MCST, Moscow Institute of Physics and Technology, Moscow State University, Orenburg State University, Politecnico di Milano, Rostov State University of Civil Engineering, Saint-Petersburg State Polytechnic University, Saint-Petersburg State University, Southern Federal University, State Research Institute of Aviation Systems, sysTeam GmbH, Tomsk State University, Ulyanovsk Technical State University, University of Passau, University of Southern Denmark, Volga Region State University of Telecommunication and Informatics (6 countries, 16 cities, and 25 organizations).

We would like to thank all of the participants of SYRCoSE 2016 and their advisors for interesting papers. We are also very grateful to the PC members and the external referees for their hard work on reviewing the papers and selecting the program. Our thanks go to the invited speakers, Dirk Beyer (University of Passau), Alexey Khoroshilov (ISP RAS), and Vartan Padaryan (ISP RAS). We would also like to thank our sponsors and supporters: Russian Foundation for Basic Research (grant 16-07-20256), Federal Agency of Scientific Organizations, Exactpro Systems, and EMC Research and Development Center LLC. Finally, our special thanks go to the local organizers, Eugene Kornychin (MSU) and Arif Sultanov (Recreation Center 'Krasnovidovo', MSU), for their invaluable help in organizing the colloquium at MSU's Recreation Center 'Krasnovidovo'.

Sincerely yours,

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

May 2016

# Committees

## Program Committee Chairs

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

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

## Program Committee

 Jean-Michel Adam – France  
*Pierre Mendès France University*

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

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

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

 Svetlana I. Chuprina – Russia  
*Perm State National Research University*

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

 Liliya Yu. Emaletdinova – Russia  
*Kazan National Research Technical University*

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

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

 Maxim L. Gromov – Russia  
*Tomsk State University*

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

 Shihong Huang – USA  
*Florida Atlantic University*

 Iosif L. Itkin – Russia  
*Exactpro Systems*

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

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

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

 Alexander N. Kovartsev – Russia  
*Samara State Aerospace University*

 Vladimir P. Kozыrev – Russia  
*National Research Nuclear University "MEPhI"*

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

 Peter G. Larsen – Denmark  
*Aarhus University*

 Roustam H. Latypov – Russia  
*Kazan Federal University*

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

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

 Alexander V. Lipanov – Ukraine  
*Kharkov National University of Radioelectronics*

 Irina A. Lomazova – Russia  
*Higher School of Economics*

 Lyudmila N. Lyadova – Russia  
*Higher School of Economics*

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

 Victor M. Malyshko – Russia  
*Moscow State University*

 Tiziana Margaria – Germany  
*Lero – The Irish Software Research Centre*

 Manuel Mazzara – Russia  
*Innopolis University*

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

 Alexander S. Mikhaylov – Russia  
*RN-Inform*

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

 Alexey M. Namestnikov – Russia  
*Ulyanovsk State Technical University*

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

 Mykola S. Nikitchenko – Ukraine  
*Kyiv National Taras Shevchenko University*

 Sergey P. Orlov – Russia  
*Samara State Technical University*

 Elena A. Pavlova – Russia  
*Microsoft*

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

 Vladimir Yu. Popov – Russia  
*Ural Federal University*

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

 Rustam A. Sabitov – Russia  
*Kazan National Research Technical University*

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

 Ruslan L. Smelyansky – Russia  
*Moscow State University*

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

 Petr I. Sosnin – Russia  
*Ulyanovsk State Technical University*

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

 Andrei N. Tiugashev – Russia  
*Samara State Aerospace University*

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


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

 Dmitry Yu. Volkanov – Russia  
*Moscow State University*

 Mikhail V. Volkov – Russia  
*Ural Federal University*

 Nadezhda G. Yarushkina – Russia  
*Ulyanovsk State Technical University*


 Rostislav Yavorsky – Russia  
*Higher School of Economics*


 Nina V. Yevtushenko – Russia  
*Tomsk State University*

 Vladimir A. Zakharov – Russia  
*Moscow State University*

 Sergey S. Zaydullin – Russia  
*Kazan National Research Technical University*

## Organizing Committee

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

 Eugene V. Kornychin – Russia  
*Moscow State University*

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

## Referees

Ivan Andrianov

Nikita Astrakhantsev

Sergey Avdoshin

Nadezhda Bahareva

Oleg Borisenko

Mikhail Chupilko

Kyrylo Chykhhradze

Pavel Drobintsev

Misha Drobyshevskii

Mohamed Elwakil

Victor Gergel

Andrey Gomzin

Efim Grinkrug

Maxim Gromov

Shihong Huang

Iosif Itkin

Leonard Johard

Alexander Kamkin

Mansur Khazeev

Andrei Klimov

Anton Korshunov

Artem Kotsynyak

Alexander Kovartsev

Ilya Kozlov

Vladimir Kozyrev

Natalia Kushik

Peter Gorm Larsen

Roustam Latypov

Mikhail Lebedev

Irina Lomazova

Jorge Lopez

Lyudmila Lyadova

Victor Malyshko

Vladimir Mayorov

Manuel Mazzara

Alexander Mikhaylov

Alexey Namestnikov

Yaroslav Nedumov

Mykola Nikitchenko

Sergey Orlov

Alexander Petrenko

Ivan Piletski

Alexander Protsenko

Delhibabu Radhakrishnan

Nikolay Shilov

Kirill Skorniakov

Sergey Smolov

Valeriy Sokolov

Petr Sosnin

Veniamin Tarasov

Andrei Tatarnikov

Alexander Tchitchigin

Andrei Tiugashev

Denis Turdakov

Maksim Varlamov

Dmitry Volkanov

Mikhail Volkov

Rostislav Yavorskiy

Nina Yevtushenko

Vladimir Zakharov

Mark Zhitomirski

# Language Support for Generic Programming in Object-Oriented Languages: Design Challenges

Julia Belyakova

Institute for Mathematics, Mechanics  
and Computer Science  
named after I.I. Vorovich  
Southern Federal University  
Rostov-on-Don, Russia  
Email: julbel@sfedu.ru

**Abstract**—It is generally considered that object-oriented (OO) languages provide weaker support for generic programming (GP) as compared with such functional languages as Haskell or SML. There were several comparative studies which showed this. But many new object-oriented languages have appeared in recent years. Have they improved the support for generic programming? And if not, is there a reason why OO languages yield to functional ones in this respect? In the earlier comparative studies object-oriented languages were usually not treated in any special way. However, the OO features affect language facilities for GP and a style people write generic programs in such languages. In this paper we compare ten modern object-oriented languages and language extensions with respect to their support for generic programming. It has been discovered that every of these languages strictly follows one of the two approaches to constraining type parameters. So the first design challenge we consider is “which approach is better”. It turns out that most of the explored OO languages use the less powerful one. The second thing that has a big impact on the expressive power of a programming language is support for multiple models. We discuss pros and cons of this feature and its relation to other language facilities for generic programming.

## I. INTRODUCTION

Almost all modern programming languages provide language support for generic programming (GP) [1]. Some languages do it better than others. For example, Haskell is generally considered to be one of the best languages for generic programming [2, 3], whereas such mainstream object-oriented languages as C# and Java are much less expressive and have many drawbacks. There were several studies that compared language support for generic programming in different languages [2–5]. However, these studies do not make any difference between object-oriented and functional languages. We argue that OO languages are to be treated separately, because they support the distinctive OO features that pure functional languages do not, such as inheritance, interfaces/traits, subtype polymorphism, etc. These features affect the language design and a way people write generic programs in object-oriented languages.

Several new object-oriented languages have appeared in recent years, for instance, Rust, Swift, Kotlin. At the same time, several independent extensions have been developed for mainstream OO languages [6–9]. These new languages and

extensions have many differences, but all of them tend to improve the support for generic programming. There is a lack of a careful comparison of the approaches and mechanisms for generic programming in *modern object-oriented* languages. This study is aimed to fill the gap: it gives a survey, analysis, and comparison of the facilities for generic programming that the chosen OO languages provide. We identify the dependencies between major language features, detect incompatible ones, and point the properties that a language design should satisfy to be effective for generic programming.

## II. MAIN IDEAS

Ten modern object-oriented languages and language extensions have been explored in this study with respect to generic programming. We have found out that in the case of OO languages there are exactly two approaches to the design of language constructs for generic programming. We call the first one “constraints-are-types”, because under this approach such OO constructs as interfaces or traits, which are usually used as types in object-oriented programs, are also used to constrain type parameters in generic programs. The second approach, “constraints-are-Not-types”, restricts OO constructs to be used as types only, and provides separate language constructs for constraining type parameters. Hence the first design challenge arises: is one of this approaches better than another? Or the same expressive power can be achieved using any of them? We answer these questions in [Sec. III](#). It turns out that the approaches cannot be integrated together, and the second one is more expressive.

The second point covered in the paper in detail (in [Sec. IV](#)) is language support for multiple models (by “model” we mean a way in which types satisfy constraints). There are several questions related to multiple models:

- 1) Is it desirable to have multiple models of a constraint?
- 2) How can support for multiple models be provided with the approaches we have discovered?
- 3) Why does not Haskell allow multiple models (instances of a type class)?
- 4) Is there a language design that reflects the support for multiple models better than the existing ones?

The short answers are:

---

```

interface IPrintable { string Print(); }

void PrintArr(IPrintable[] xs)
{ foreach (var x in xs)
  Console.WriteLine("{0}\n", x.Print()); }

string InParens<T>(T x) where T : IPrintable
{ return "(" + x.Print() + ")"; }

```

---

Fig. 1. An ambiguous role of C# interfaces

- 1) Yes, it is desirable.
- 2) It can be naturally provided with the second approach but not with the first one.
- 3) Because of type inference.
- 4) Yes, there is.

In conclusion, we present a modified version of the well-known table [2, 4] showing the levels of language support for the features important for generic programming. Table I provides information on all of the object-oriented languages considered, introduces some new features, and demonstrates the relations between the features.

### III. TWO APPROACHES TO CONSTRAINING TYPE PARAMETERS

This section provides a survey of *language constructs for generic programming* in several modern *object-oriented* programming languages as well as some language extensions. All of the languages we explored adopt one of the two approaches:

- 1) Interface-like constructs, which are normally used as types in object-oriented programming, are also used to constrain type parameters. By “interface-like constructs” we mean, in particular, C#/Java interfaces, Scala traits, Swift protocols, Rust traits. Fig. 1 shows a corresponding example in C#: `IPrintable` interface acts as the type of `xs` in `PrintArr`, whereas in the function `InParens<T>` it is used to constrain the type parameter `T`.
- 2) For constraining type parameters a separate language construct is provided; such construct cannot be used as a type. We will see some examples in Sec. III-B.

Sec. III-A analyses the languages of the first category; Sec. III-B is devoted to the second one. In Sec. III-C we compare both approaches and answer the question “Which one is better if any?”.

#### A. Languages with “Constraints-are-Types” Philosophy

C# and Java are probably the best-known programming languages in this category. Note that an interface (or a similar language construct) describes properties, an interface of a *single* type that implements/extends it. This has inevitable consequence: *multi-type constraints* (constraints on several types) cannot be expressed naturally. Consider a generic unification algorithm [10]: it takes a set of equations between terms (symbolic expressions), and returns the most general substitution which solves the equations. So the algorithm operates on three kinds of data: terms, equations, substitutions. A signature of the algorithm might be as follows:

```

Substitution Unify<Term, Equation, Substitution>
  (IEnumerable<Equation>)

```

---

```

interface ITerm<Tm> { IEnumerable<Tm> Subterms(); ... }

interface IEquation<Tm, Eqtn, Subst>
  where Tm : ITerm<Tm>
  where Eqtn : IEquation<Tm, Eqtn, Subst>
  where Subst : ISubstitution<Tm, Eqtn, Subst>
{ Subst Solve();
  IEnumerable<Eqtn> Split(); ... }

interface ISubstitution<Tm, Eqtn, Subst>
  where Tm : ITerm<Tm>
  where Eqtn : IEquation<Tm, Eqtn, Subst>
  where Subst : ISubstitution<Tm, Eqtn, Subst>
{ Tm SubstituteTm(Tm);
  IEnumerable<Eqtn> SubstituteEq (IEnumerable<Eqtn>); ... }

```

---

Fig. 2. The C# interfaces for unification algorithm

---

```

interface IComparable<T> { int CompareTo(T other); }

class SortedSet<T> where T : IComparable<T> {...}

```

---

Fig. 3. The `IComparable<T>` interface in C#

But a bunch of functions has to be provided to implement the algorithm: `Subterms : Term → IEnumerable<Term>`, `Solve : Equation → Substitution`, `SubstituteTm : Substitution × Term → Term`, `SubstituteEq : Substitution × IEnumerable<Equation> → IEnumerable<Equation>`, and some others. All these functions are needed for unification at once, hence it would be convenient to have a single constraint that relates all the type parameters and provides the functions required.

```

Substitution Unify<Term, Equation, Substitution>
  (IEnumerable<Equation>) where <single constraint>

```

But in C#/Java the only thing one can do<sup>1</sup> is to define three different interfaces for `Term`, `Equation`, and `Substitution`, and then separately constrain every type parameter with a respective interface. Fig. 2 shows the interface definitions. To set up a relation between mutually dependent interfaces, three type parameters are used: `Tm` for terms, `Eqtn` for equations, and `Subst` for substitution. Moreover, the parameters are repeatedly constrained with the appropriate interface in every interface definition. That constraints are to be stated in a signature of the unification algorithm as well:

```

Subst Unify<Tm, Eqtn, Subst> (IEnumerable<Eqtn>)
  where Tm : ITerm<Tm>
  where Eqtn : IEquation<Tm, Eqtn, Subst>
  where Subst : ISubstitution<Tm, Eqtn, Subst>

```

There is one more thing to notice here — interfaces are used in both roles in the same piece of code: the `IEnumerable<Eqtn>` interface is used as a type, whereas other interfaces in the where sections are used as constraints.

The problem of multi-type constraints is a common thing for OO languages in the first category, but C# and Java have various drawbacks besides that [2, 8]. In comparison with other programming languages that support generic programming (not only object-oriented), these are much less expressive. An incomplete list of drawbacks is enumerated below.

<sup>1</sup>The Concept design pattern can also be used, but it has its own drawbacks. We will discuss concept pattern later, in Sec. IV-C2.



- *Lack of retroactive interface implementation.* After the type had been defined, it cannot implement any new interface. A consequence is that a generic code with constraints on type parameters can only be instantiated with types *originally* designed to satisfy these constraints. It is impossible to adapt the type afterwards, even if it semantically conforms the constraints.
- *Drawbacks of F-bounded polymorphism.* F-bounded polymorphism [11] allows “recursive” constraints (F-constraints) on type parameters in the form  $T : I<T>$ , where  $T$  is a type parameter,  $I<>$  is a generic interface. Such kind of constraints solves the binary method problem [12]: Fig. 3 demonstrates a corresponding C# [13] example. The type parameter  $T$  in the interface  $IComparable<T>$  pretends to be a type that implements this interface. This is indeed the case for the class  $SortedSet<T>$  due to the constraint  $T : IComparable<T>$ , so the method  $T.CompareTo(T)$  is like a binary function for comparing elements of type  $T$ . But the semantics of  $IComparable<T>$  itself has nothing to do with binary methods. One could easily write some class  $Foo$  implementing  $IComparable<Bar>$ , and thus the semantics of comparing two  $Bars$  would be broken. Another shortcoming of F-bounded polymorphism is that a code with recursive constraints is rather cumbersome and difficult to understand. Yet, as we will see, F-bounded polymorphism is not the only solution for the binary method problem. More detailed discussion on pitfalls of F-bounded polymorphism can be found in [8] and [14].
- *Lack of associated types* [14, 15]. Types that are logically related to some entity are often called associated types of the entity. For instance, types of edges and vertices are associated types of a graph. There is no specific language support for associated types in C# and Java: such types are expressed in generic code in the form of extra type parameters.
- *Lack of constraints propagation* [14, 15]. Look at the following code:

```
void baz<T>(SortedSet<T> s)
    where T : IComparable<T> { ... }
```

The function  $baz<T>$  takes a value of the type  $SortedSet<T>$ ; in the definition of  $SortedSet<T>$  in Fig. 3 the type parameter  $T$ , type of elements, is constrained with  $IComparable<T>$ . In the  $baz<T>$  definition  $T$  has to be also constrained, otherwise the code would not compile: a compiler does not propagate the constraints implied by formal parameters, that is a programmer’s burden.

Some of these drawbacks were eliminated in modern object-oriented languages. In the following subsections we briefly examine language facilities for generic programming in the modern OO languages with “constraint-are-types” philosophy.

1) *Interfaces in Ceylon and Kotlin:* In contrast to C#, Ceylon [16] and Kotlin [17] interfaces support *default method implementation*, so Java 8 [18] interfaces do. This is a useful feature for generic programming. For instance, one

```
interface Equatable<T> {
    fun equal (other: T) : Boolean
    fun notEqual (other: T): Boolean
    { return !this.equal(other) }
}

class Ident (name : String) : Equatable<Ident> {
    val idname = name.toUpperCase()
    override fun equal (other: Ident) : Boolean
    { return idname == other.idname }
```

Fig. 4. Interfaces and constraints in Kotlin

```
shared interface Comparable<Other> of Other
    given Other satisfies Comparable<Other> {
        shared formal Integer compareTo(Other other);
        shared Integer reverseCompareTo(Other other) {
            return other.compareTo(this);
        } }
```

Fig. 5. The use of “self type” in Ceylon interfaces

```
struct Point { x: i32, y: i32, }
...
impl Point {
    fn moveOn(&self, dx: i32, dy: i32) -> Point
    { Point {x: self.x + dx, y: self.y + dy } }
    ...
    impl Point {
        fn reflect(&self) -> Point
        { Point {x: -self.x, y: -self.y } }
    }
    ...
    let p1 = Point {x: 4, y: 3};
    let p2 = p1.moveOn(1, 1);    let p3 = p1.reflect();
```

Fig. 6. Point struct and its methods in Rust

can define an interface for equality that provides a default implementation for inequality operation. Fig. 4 demonstrates corresponding Kotlin definitions: the  $Ident$  class implements the interface  $Equatable<Ident>$  that has two methods,  $equal$  and  $notEqual$ ; as long as  $notEqual$  has a default implementation in the interface, there is no need to implement it in the  $Ident$  class. In addition to default method implementations, the Ceylon language also allows to declare a type parameter as a *self type*. An example is shown in Fig. 5. In the definition of the  $Comparable<Other>$  interface the declaration of  $Other$  explicitly requires  $Other$  to be a self type of the interface, i.e. a type that implements this interface. Because of this the  $reverseCompareTo$  method can be defined: the  $other$  and this values have the type  $Other$ , with the  $Other$  implementing  $Comparable<Other>$ , so the call  $other.compareTo(this)$  is perfectly legal.

2) *Scala Traits:* Similarly to advanced interfaces in Java 8, Ceylon, and Kotlin, Scala traits [5, 19] support *default method implementations*. They can also have *abstract type members*, which, in particular, can be used as *associated types* [20]. Just as in C#/Java/Ceylon/Kotlin, type parameters (and abstract types) in Scala can be constrained with traits and supertypes (upper bounds): the latter constraints are called *subtype constraints*. But, moreover, they can be constrained with subtypes (lower bounds), which is called *supertype constraints* respectively. None of the languages we discussed so far support supertype constraints nor associated types. Another important Scala feature, implicits [19], will be mentioned later in Sec. IV-A with respect to the Concept design pattern.

```

trait Eqtbl { fn equal(&self, that: &Self) -> bool;
    fn not_equal(&self, that: &Self) -> bool
    { !self.equal(that) }}
trait Printable { fn print(&self); }
...
impl Eqtbl for i32 {
    fn equal (&self, that: &i32) -> bool { *self == *that }
    ...
struct Pair<S, T>{ fst: S, snd: T }
...
impl <S : Eqtbl, T : Eqtbl> Eqtbl for Pair<S, T> {
    fn equal (&self, that: &Pair<S, T>) -> bool
    { self.fst.equal(&that.fst) && self.snd.equal(&that.snd) }}

```

Fig. 7. An example of using Rust traits

3) *Rust Traits*: Rust language [21] quite differs from other object-oriented languages. There is no traditional class construct in Rust, but instead it suggests structs that store the data, and separate method implementations for structs. An example is shown in Fig. 6<sup>2</sup>: two `impl Point` blocks define method implementations for the `Point` struct. If a function takes the `&self`<sup>3</sup> argument (as `moveOn`), it is treated as a method. There can be any number of implementation blocks, yet they can be defined at any point after the struct declaration (even in a different module). This gives a huge advantage with respect to generic programming: any struct can be *retroactively* adapted to satisfy constraints.

Constraints in Rust are expressed using traits. A trait defines which methods have to be implemented by a type similarly to Scala traits, Java 8 interfaces, and others. Traits can have *default method implementations* and *associated types*; besides that, a *self type* of the trait is directly available and can be used in method definitions. Fig. 7<sup>4</sup> demonstrates an example: the `Eqtbl` trait defining equality and inequality operations. Note how support for self type solves the binary method problem (here `equal` is a binary method): there is no need in extra type parameter that “pretends” to be a self type, because the self type `Self` is already available.

Method implementations in Rust can be probably thought of similarly to .NET “extension methods”. But in contrast to .NET<sup>5</sup>, types in Rust also can *retroactively implement traits* in `impl` blocks as shown in Fig. 7: `Eqtbl` is implemented by `i32` and `Pair<S, T>`. The latter definition also demonstrates a so-called *type-conditional implementation*: pairs are equality comparable only if their elements are equality comparable. The constraint `<S : Eqtbl...>` is a shorthand, it can be declared in a `where` section as well.

There is no struct inheritance and subtype polymorphism in Rust. Nevertheless, as long as traits can be used not only as constraints but also as types, a dynamic dispatch is provided through a feature called trait objects. Suppose `i32` and `f64`

<sup>2</sup>Some details were omitted for simplicity. To make the code correct, one has to add `#[derive(Debug, Copy, Clone)]` before the `Point` definition.

<sup>3</sup>The “&” symbol means that an argument is passed by reference.

<sup>4</sup>Some details were omitted for simplicity. The following declaration is to be provided to make the code correct: `#[derive(Copy, Clone)]` before the definition `struct Pair<S : Copy, T : Copy>`. Yet the type parameters of the `impl` for `pair` must be constrained with `Copy+Equtable`.

<sup>5</sup>Similarly to .NET, Kotlin supports extending classes with methods and properties, but interface implementation in extensions is not allowed.

```

protocol Equatable { func equal(that: Self) -> Bool; }
extension Equatable { func notEqual(that: Self) -> Bool
    { return !self.equal(that) }}
func contains<T : Equatable>
    (values: [T], x:T) -> Bool { ... }

protocol Printable { func print(); }
extension Int : Printable { ... }

protocol Container { associatedtype ItemTy ... }
func allItemsMatch<C1: Container, C2: Container>
    where C1.ItemTy == C2.ItemTy, C1.ItemTy: Equatable> ...

```

Fig. 8. Protocols and their use in Swift

implement the `Printable` trait from Fig. 7. Then the following code demonstrates creating and use of a polymorphic collection (the type of the `polyVec` elements is a reference type):

```

let pr1 = 3; let pr2 = 4.5; let pr3 = -10;
let polyVec: Vec<&Printable> = vec! [&pr1, &pr2, &pr3];
for v in polyVec { v.print(); }

```

4) *Swift Protocols*: Swift is a more conventional OO language than Rust: it has classes, inheritance, and subtype polymorphism. Classes can be extended with new methods using extensions that are quite similar to Rust method implementations. Instead of interfaces and traits Swift provides protocols. They cannot be generic but support *associated types* and *same-type* constraints, *default method implementations* through protocol extensions, and explicit access to the *self type*; due to the mechanism of extensions, types can *retroactively* adopt protocols. Fig. 8 illustrates some examples: the `Equatable` protocol extended with a default implementation for `notEqual` (pay attention to the use of the `Self` type); the `contains<T>` generic function with a protocol constraint on the type parameter `T`; an extension of the type `Int` that enables its conformance to the `Printable` protocol; the `Container` protocol with the associated type `ItemTy`; the `allItemsMatch` generic function with the same-type constraint on types of elements of two containers, `C1` and `C2`.

## B. Languages with “Constraints-are-Not-Types” Philosophy

Most of the languages in this category were to some extent inspired by the design of Haskell type classes [22]. For defining constraints these languages suggest *new language constructs*, which are usually second-class citizens<sup>6</sup>. These constructs have *no self types* and *cannot* be used as types, they describe requirements on type parameters in external way; therefore, retroactive constraints satisfaction (*retroactive modeling*) is automatically provided. Besides retroactive modeling, an integral advantage of such kind of constructs is that *multi-type constraints* can be easily and naturally expressed using them; yet there is no semantic ambiguity which arises when the same construct, such as C# interface, is used both as a type and constraint, as in the example below:

```

void Sort<T>(ICollection<T>) where T : IComparable<T>;

```

Here `ICollection<T>` and `IComparable<T>` are generic interfaces, but the former is used as a type whereas the latter is used as constraint.

<sup>6</sup>Second-class citizens cannot be assigned to variables, passed as arguments, returned from functions.



```

interface EQ { boolean eq(This that);
               boolean notEq(This that); }
abstract implementation EQ [EQ] {
    boolean notEq(This that) { return !this.eq(that); }}

boolean contains<X>(List<X> list, X x)
    where X implements EQ { ... }

abstract class Expr {...}    class IntLit extends Expr {...}
class PlusExpr extends Expr { Expr left; Expr right; ... }
...
implementation EQ [Expr] {
    boolean eq(Expr that) { return false; }}
implementation EQ [PlusExpr] {boolean eq(PlusExpr that){...}}

interface UNIFY [Tm, Eqtn, Subst] {
    receiver Tm    { IEnumerable<Tm> Subterms(); ... }
    receiver Eqtn  { IEnumerable<Eqtn> Split(); ... }
    receiver Subst { Tm SubstituteTm(Tm); ... }
    Subst Unify<Tm, Eqtn, Subst>(Enumerable<Eqtn>)
    where [Tm, Eqtn, Subst] implements UNIFY {...}

```

Fig. 9. Generalized interfaces in JavaGI

1) *JavaGI Generalized Interfaces*: JavaGI [6] generalized interfaces represent a kind of confluence of both “constraints-are-types” and “constraints-are-not-types” philosophies. Such interfaces as `PrettyPrintable` defined below are called single-parameter interfaces. They describe an interface of a single type and can be used both as types and constraints.

```

interface PrettyPrintable { String prettyPrint(); }

```

Such interfaces have explicit access to the *self type* named `This`; an example is shown in Fig. 9, where the self type is used in the interface `EQ`. There is no direct support for default method implementations in JavaGI, but *abstract implementation definitions* can be used for this purpose<sup>7</sup>. For example, the `notEq` method of `EQ` (Fig. 9) is implemented in such a way. Generalized interfaces can be implemented *retroactively* in *implementation blocks*. They do not support associated types but can be generic; moreover, implementations can be generic as well, and support for *type-conditional interface implementation* is provided:

```

implementation<S, T> EQ [Pair<S, T>] where S implements EQ
    where T implements EQ { ... }

```

Besides single-parameter interfaces, there are *multi-headed* generalized interfaces that adopt several features from Haskell type classes [23] and describe interfaces of several types. There is no self type in a multi-headed interface; therefore, it cannot be used as a type, it is designed to be used as a constraint *only*. An example of multi-headed interface is shown in Fig. 9: the `UNIFY` interface contains all the functions required by the unification algorithm considered earlier; the requirements on three types (term, equation, substitution) are defined at once in a single interface. Note how succinct is this definition as compared with the one in Fig. 2.

2) *Language G and C++ concepts*: Concept as an explicit language construct for defining constraints on type parameters was initially introduced in 2003 [24]. Several designs have

<sup>7</sup>The design of JavaGI we discuss here goes back to 2011 when default method implementations were not supported in Java. With Java 8 this task could probably be solved in a more elegant way.

```

concept InputIterator<Iter> { type value; ... }
concept Monoid<T> { fun identity_elt() -> T;
                  fun binary_op(T, T) -> T; }

model Monoid<int>
{ fun identity_elt() -> int@ { return 0; } ... };

fun accumulate<Iter> where { InputIterator<Iter>,
                             Monoid<InputIterator<Iter>.value> }
(Iter first, Iter last) -> InputIterator<Iter>.value
{ let init = identity_elt(); ... }

```

Fig. 10. Concepts and their use in G

been developed since that time [25–27]; in the large, the expressive power of concepts is rather close the Haskell type classes [3]. Concepts were to solve the problems of unconstrained C++ templates [14, 28]; they were expected to be included in C++0x standard, but this did not happen. A new version of concepts, Concepts Lite (C++1z) [29], is under way now. The language G declared as “a language for generic programming” [7] also provides concepts that are very similar to the C++0x concepts. G is a subset of C++ extended with several constructs for generic programming. For “C++ concepts” we use the G syntax in this paper.

Similarly to a type class, a concept defines a set of requirements on one or more type parameters. It can contain *function signatures* that may be accompanied with *default implementations*, *associated types*, nested *concept-requirements* on associated types, and *same-type constraints*. A concept can *refine* one or more concepts, it means that refining concept includes all the requirements from the refined concepts. Refinement is very similar to multiple interface inheritance in C# or protocol inheritance in Swift. Due to the concept refinement, a so-called *concept-based overloading* is supported: one can define several versions of an algorithm/class that have different constraints, and then at compile time the most specialized version is chosen for the given instance. The C++ *advance* algorithm for iterators is a classic example of concept-based overloading application.

It is said that a type (or a set of types) *satisfies* a concept if an appropriate model of the concept is defined for this type (types). Model definitions are independent from type definitions, so the modeling relation is established *retroactively*; models can be generic and *type-conditional*. Fig. 10 illustrates some examples: the `InputIterator<Iter>` concept with the associated type of elements `value`; the `Monoid<T>` concept and its model for the type `int`; the `accumulate<Iter>` generic function with two constraints, on the type of an iterator and on the associated type of this iterator. Note how `identity_elt` is called in `accumulate`: in contrast to the languages from the previous section, `identity_elt` is available in the body of `accumulate` at the top-level; this may lead to some inconvenience even if the autocomplete feature is supported in IDE.

3) *C# with concepts*: In the C#<sup>cp</sup> project [8] (C# with concepts) concept mechanism integrates with subtyping: type parameters and associated types can be constrained with *super-types* (as in basic C#) and also with *subtypes* (as in Scala). In contrast to all of the languages we discussed earlier, C#<sup>cp</sup> allows *multiple models* of a concept in the same scope. Some ex-

---

```

concept CEquatable[T] { bool Equal(T x, T y);
    bool NotEqual(T x, T y) { return !Equal(x, y); }}

interface ISet<T> where CEquatable[T] { ... }

model default StringEqCaseS for CEquatable[String] { ... }
model StringEqCaseIS for CEquatable[String] { ... }

bool Contains<T>(IEnumerable<T> values, T x)
    where CEquatable[T] using CEq {... if (cEq.Equal(...) ...)}

```

---

Fig. 11. Concepts and models in C#<sup>cp1</sup>

---

```

constraint Eq[T] { boolean T.equals(T other); }
constraint GraphLike[V, E] { V E.source(); ... }

interface Set[T where Eq[T]] { ... }

model CIEq for Eq[String] { ... } // case-insensitive model

model DualGraph[V,E] for GraphLike[V,E]
    where GraphLike[V,E] g
{ V E.source() { return this.(g.sink)(); } ... }

```

---

Fig. 12. Constraints and models in Genus

amples are shown in Fig. 11: the `CEquatable[T]` concept with the `Equal` signature and a default implementation of `NotEqual`, the generic interface `ISet<T>` with concept-requirement on the type parameter `T`, and two models of `CEquatable[]` for the type `String` — for case-sensitive and case-insensitive equality comparison. The first model is marked as a *default* model<sup>8</sup>: it means that this model is used if a model is not specified at the point of instantiation. For instance, in the following code `StringEqCaseS` is used to test strings equality in `s1`.

```

ISet<String> s1 = ...;
ISet<String> [using StringEqCaseIS] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types

```

Note that `s1` and `s2` have different types because they use different models of `CEquatable[String]`. This property is called “constraints-compatibility” in [8], but we will refer to it as “models-consistency”. One more interesting thing about C#<sup>cp1</sup>: concept-requirements can be named. In the `Contains<T>` function (Fig. 11) the name `cEq` is given to the requirement on `T`; this name is used later in the body of `Contains<T>` to access the `Equal` function of the concept. It is also worth mention that the interface `IEnumerable<T>` is used as a type along with the concept `CEquatable[T]` being used as a constraint; thus, the role of interfaces is not ambiguous any more, interfaces and concepts are independently used for different purposes.

4) *Constraints in Genus*: Like G concepts and Haskell type classes, constraints in Genus [9] (an extension for Java) are used as constraints only. Fig. 12 demonstrates some examples: the `Eq[T]` constraint, which is used to constrain the `T` in the `Set[T]` interface; the model of `Eq[String]` for case-insensitive equality comparison; the multi-parameter constraint `GraphLike[V, E]`, and the type-conditional generic model `DualGraph[V,E]`. Methods in Genus classes/interfaces can impose additional constraints:

<sup>8</sup>The default model can be generated automatically for a type if the type conforms to a concept, i.e. it provides methods required by the concept.

---

```

interface List[E] { boolean remove(E e) where Eq[E]; ... }

```

Here the `List[]` interface can be instantiated by any type, but the `remove` method can be used only if the type `E` of elements satisfies the `Eq[E]` constraint. This feature is called *model genericity*.

Just as C#<sup>cp1</sup>, Genus supports *multiple models* and automatic generation of the *natural* model, which is the same thing as the default model in C#<sup>cp1</sup>. Due to this, the following code causes a static type error (we saw the same example in C#<sup>cp1</sup>):

```

Set[String] s1 = ...;
Set[String with CIEq] s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types

```

In Genus this feature is called *model-dependent types*. An important note is to be made here: in contrast to true dependent types that depend on *values*, model-dependent types depend on models, which are compile-time artefacts. So the model-dependent types are just as dependent as generic types are type-dependent types.

As well as concept-requirements in C#<sup>cp1</sup>, constraint-requirements in Genus can be named; the example is shown in Fig. 12: `g` is a name of the `GraphLike[V,E]` constraint required by the `DualGraph[V,E]` model. Because function signatures inside constraints are declared with an explicit receiver type (in a style close to JavaGI), such as the type `T` in the `Eq[T]` constraint, syntax of calls to functions in the case of named models is `_.(g.sink)()`, not `g.sink(_)`.

### C. Which Philosophy Is Better If Any?

It is time to find out which approach is better. Taking into consideration what we explored in Sec. III-A and Sec. III-B, we draw a conclusion that there are only two language features that cannot be incorporated in a language *together*:

- 1) the use of a construct both as a type and constraint;
- 2) natural support for multi-type constraints.

Languages with “constraints-are-types” philosophy support the first feature but not the second, languages with “constraints-are-Not-types” philosophy vice versa<sup>9</sup>. Can we determine one feature that is more important?

It was shown in the study [30] that in practice interfaces that are used as constraints (such as `IComparable<T>` in C# or `Comparable<X>` in Java) are almost never used as types:

<sup>9</sup>JavaGI seems to support both of them, but it actually provides different constructs for different purposes: single-parameter interfaces are more like Rust traits or Swift protocols, whereas multi-headed interfaces are similar to concepts and type classes; the latter cannot be used as types.

authors had checked about 14 millions lines of Java code and found only one such example, and, furthermore, it was rewritten and eliminated. It is also mentioned in [30] that the same observation holds for the code in Ceylon.

It is hard to imagine any useful “constraint-and-type” example besides the `IPrintable` interface from Fig. 1. In those rare cases when this could happen, it is possible to provide a lightweight language mechanism for automatic generation of one construct from another. For example, single-parameter Genus constraints with some restrictions could be translated to Java interfaces, with the other direction being easier. At the same time, multi-type constraints, which can be so naturally expressed under the “constraints-are-Not-types” approach, have rather awkward and cumbersome representation in the “constraints-are-types” approach. All other language facilities we discussed could be supported under any approach. Therefore, we claim that the “constraints-are-Not-types” approach is preferable. An additional benefit is that it eliminates the ambiguity in semantics of the interface-like constructs.

#### IV. SINGLE MODEL VERSUS MULTIPLE MODELS

For simplicity, in this part of the paper we call “constraint” any language construct that is used to describe constraints, while a way in which types satisfy the constraints we call “model”. We have seen in the previous section that most of the languages allow to have only one, unique model of a constraint for the given set of types; only C<sup>#pt</sup> [8] and Genus [9] support multiple models<sup>10</sup>. And indeed this makes sense for the languages with “constraints-are-types” philosophy, because it is not clear what to do with types that could implement interfaces (or any other similar constructs) in several ways. But how does this affect generic programming?

It turns out that sometimes it is desirable to have multiple models of a constraint for the same set of types. The example of string sets with case-sensitive and case-insensitive equalities we saw earlier is one of such examples; another one is the use of different orderings, yet different graph implementations, and so on. Thus, in respect of generic programming, the absence of multiple models is rather a problem than a benefit. Without extending the language the problem of multiple models can be solved in two ways:

- 1) Using the Adapter pattern. If one wants the type `Foo` to implement `IComparable<Foo>` in a different way, an adapter of `Foo`, the `Foo1` that implements `IComparable<Foo1>` can be created. This adapter then can be used instead of `Foo` whenever the `Foo1`-style comparison is required. An obvious shortcoming of this approach is the need to repeatedly wrap and unwrap `Foo` values; in addition, a code becomes cumbersome.
- 2) Using the Concept pattern, which is considered in Sec. IV-A.

Both approaches have serious drawbacks. Moreover, as we have discovered in Sec. III-C, languages with the “constraints-are-types” philosophy are in the large less expressive than ones

---

```
// F-bounded polymorphism
interface IComparable<T> { int CompareTo(T other); }
void Sort<T>(T[] values) where T : IComparable<T> { ... }
class SortedSet<T> where T : IComparable<T> { ... }

// Concept Pattern
interface IComparer<T> { int Compare(T x, T y); }
void Sort<T>(T[] values, IComparer<T> cmp) { ... }
class SortedSet<T> { private IComparer<T> cmp; ...
public SortedSet(IComparer<T> cmp) { ... } ... }
```

---

Fig. 13. The use of the Concept design pattern in C#

with the “constraints-are-Not-types” philosophy. But may such languages as C<sup>#pt</sup> and Genus, which are in the “constraints-are-Not-types” category and support multiple models at the language level, be considered as the best languages for generic programming, or we can imagine a language with a better design? And one more question: if language support for multiple models is a good idea, then why does not Haskell [23] allow multiple instances of a type class? After all, it is considered to be one of the most expressive languages for generic programming. We answer the latter question in Sec. IV-B and discuss the former one in Sec. IV-C.

##### A. Concept Pattern

The Concept design pattern is suitable for programming languages with the “constraints-are-types” philosophy. It eliminates two problems:

- 1) First, it enables *retroactive modeling* of constraints, which is not supported in such languages as C#, Java, Ceylon, Kotlin, or Scala.
- 2) Second, it allows to define *multiple models* of a constraint for the same set of types.

The idea of the Concept pattern is as follows: instead of constraining type parameters, generic functions and classes take extra arguments that provide a required functionality — “concepts”. Fig. 13 shows an example: in the case of the Concept pattern the F-constraint `T : IComparable<T>` is replaced with an extra argument of the type `IComparer<T>`. The `IComparer<T>` interface represents a concept of comparing: it describes the interface of an object that can compare values of the type `T`. As long as one can define several classes implementing the same interface, different “models” of the `IComparer<T>` “concept” can be passed into `Sort<T>` and `SortedSet<T>`.

This pattern is widely used in generic libraries of such mainstream object-oriented languages as C# and Java; it is also used in Scala. Due to implicits [5, 19], the use of the Concept pattern in Scala is a bit easier: in most cases an appropriate “model” can be found by a compiler implicitly, so there is no need to explicitly pass it at a call site<sup>11</sup>. Nevertheless, the pattern has two substantial drawbacks. First of all, it brings *run-time overhead*, because every object of a generic class with constraints has at least one extra field for the “concept”, while generic functions with constraints take at least one

<sup>11</sup> Scala is often blamed for its complex rules of implicits resolution: sometimes it is not clear which implicit object is to be used.

<sup>10</sup>G [7] allows multiple models only in different lexical scopes.

extra argument. The second drawback, which we call *models-inconsistency*, is less obvious but may lead to very subtle errors. Suppose we have `s1` of the type `HashSet<String>` and `s2` of the *same* type, provided that `s1` uses case-sensitive equality comparison, `s2` — the case-insensitive one. Thus, `s1` and `s2` use different, inconsistent models of comparison. Now consider the following function:

```
static HashSet<T> GetUnion<T>(HashSet<T> a, HashSet<T> b)
{
    var us = new HashSet<T>(a, a.Comparer);
    us.UnionWith(b);    return us;
}
```

Unexpectedly, the result of `GetUnion(s1, s2)` could differ from the result of `GetUnion(s2, s1)`. Despite the fact that `s1` and `s2` have the same type, they use different comparers, so the result depends on which comparer was chosen to build the union. Recall that in C#<sup>pt</sup> and Genus models are part of the types; therefore, the similar situation causes a static type error. But in the case of the Concept pattern models-consistency *cannot* be checked at *compile time*.

### B. Instance Uniqueness in Haskell

Type classes in Haskell [22] provide a support for ad hoc polymorphism (function overloading). Like concepts and constraints, they define functions available for some types. For instance, a type class for equality comparison is defined as follows:

```
class Eq a where (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

It contains a function signature for equality operator `==`, and provides a default implementation for inequality operator `/=`. Then instances (models) of this type class can be defined for types. For example, an instance for `Int`, a *type-conditional* instance for lists, and so on.

```
instance Eq Int where ... -- (==) implementation
instance Eq a => Eq [a] where ... -- (==) implementation
```

As long as type classes support ad hoc polymorphism, they are “globally transparent”. If a function is a part of some type class, every time the name of this function is used a compiler knows that an instance of the corresponding type class must be provided. And there is a strong reason why multiple instances of a type class for the same set of types are not allowed in Haskell: it is *type inference*. Consider the following function definition:

```
foo xs ys = if xs == ys then xs else xs ++ ys
```

In Haskell such definition is valid and its type can be inferred. It is `Eq a => [a] -> [a] -> [a]`<sup>12</sup>. Inference succeeds, because a compiler knows the following facts: as long as `(++)` has the type `[a] -> [a] -> [a]`, `xs` and `ys` are lists; there is an instance of `Eq` for lists (`Eq a => Eq [a]`). If there were no `Eq [a]` instance available, type checking would fail.

Now suppose that multiple instances of a type class are allowed. What to do with type inference of the `foo` in this case? To check whether there is at least one instance `Eq [a]`? And what if we also have the following code:

```
class Eq a => Baz a where
    bar :: a -> Int

useBar x y = if length x > length y then bar x - bar y
            else bar y - bar x
```

If instances are uniquely defined, type checker just checks if there is an instance `Eq [a]` that implies `Baz [a]` (`x` and `y` are inferred to be lists because `length` has the type `[a] -> Int`). But if there are multiple `Eq [a]` instances, then every `Baz [a]` instance must specify which `Eq [a]` instance it uses. It can even be the case that there is a `Baz [a]` instance for one `Eq [a]`, but not for another one. Therefore, at the point of the `useBar` *definition* a compiler has no idea whether there is an error of missed instance or not, because it knows nothing about the instances that might be used in a call to `useBar`. This information is available only at the point of a *call*.

Note that even with the `OverlappingInstances` extension for Haskell, multiple models in a sense we discuss in the paper are not supported. This extension indeed allows to have several instances that match the constraints deduced for a code. But there must be only *one* instance among them that compiler can select unambiguously (according to some rules) at the point of a code *definition*. Again, not at the call site — at the point of definition. Thus, a user of the code still cannot choose between instances, an instance is already selected by a compiler. Thus, Haskell sacrifices language support for multiple models for the sake of type inference. It is a strong argument for Haskell users, but in the case of the most object-oriented programming languages, which usually do not allow to omit type annotations of function arguments as well as constraints on type parameters, there is *no need to prohibit multiple models* in OO languages.

### C. Parameters versus Predicates

So far we have discovered that languages with “constraints-are-Not-types” philosophy, if they also allow to define multiple models, may potentially provide better support for generic programming compared to other languages. We have seen only two languages with such properties, C#<sup>pt</sup> [8] and Genus [9], and there is an essential shortcoming in the design of both of them: constraints on type parameters are declared in “predicate-style” rather than “parameter-style”. For example, consider the following Genus definition [9]:

```
Map[V,W] SSSP[V,E,W] (V s)
where GraphLike[V,E], Weighted[E,W],
    OrdRing[W], Hashable[V] { ... }
```

`SSSP[V,E,W]` is a function for Dijkstras single-source shortest-path algorithm, with the `GraphLike[V,E]`, `Weighted[E,W]`, `OrdRing[W]` and `Hashable[V]` being constraints on type parameters. The constraints look as if they were predicates on types, and if they were predicates, this function would probably be well-designed. For example, in Haskell, G, C#, Java, Rust, and many other languages, where only one model of a constraint is allowed for the given set of types, constraints on type parameters are indeed predicates: types either satisfy the constraint (if they have a model that is unique) or not. But in Genus and C#<sup>pt</sup> constraints *are not predicates*, they are

<sup>12</sup> `[a]` is a type of generic list, it is a notation for `Data.List a`



actually *parameters*, as long as different models of constraints can be used. In the worst case a call to `SSSP[V,E,W]` would be as follows:

```
...pathFromX = SSSP[MyVert, MyEdge, Double
  with MyGrLike with MyEdgeDW
  with DescDOR with MyVerHash](x);
```

Whereas in the best case:

```
...pathFromX = SSSP[MyVert, MyEdge, Double](x);
```

Note that edge and weight types cannot be deduced, because they are determined by models of the constraints, not by the vertex  $x$  itself. It is easy to imagine that models of edge weighing and its ordered ring would often vary, so a call to `SSSP[V,E,W]` is likely to look like this in many cases:

```
...pathFromX = SSSP[MyVert, MyEdge, Double
  with MyEdgeDW with DescDOR](x);
```

This is not very bad but is also not good enough.

If look again at the SSSP algorithm, one could notice that it really depends on three things: a source vertex, a model of a weighed graph which this vertex belongs to, and a model of hashing. Furthermore, at the level of the SSSP signature the type  $E$  of edges does not matter, we are interested in the model of weighed graph as a whole. Taking into account this ideas, we can rewrite the SSSP in the following way:

```
constraint WeighedGraph[V,E,W]
  extends GraphLike[V,E], Weighted[E,W], OrdRing[W] {}

Map[V,W] SSSP[V,E,W](V s)
  where WeighedGraph[V,E,W], Hashable[V] { ... }
```

Then a call to SSSP also becomes better:

```
...pathFromX = SSSP[MyVert, MyEdge, Double with MyWGr](x);
```

Nevertheless, we believe that in the case of multiple models the “predicate-style” of constraints is misleading and makes it more difficult to write and call a generic code. We suggest that the design of constraints has to be in the “parameter-style”. One example of such design is provided by the extension for the OCaml language — *modular implicits* [31]; it is briefly discussed in [Sec.IV-C1](#). A sketch of the “parameter-style” design of constraints for object-oriented languages is presented in [Sec.IV-C2](#).

1) *Modular Implicits in OCaml*: In the “modular implicits” extension for the OCaml language [31] module types are used to describe constraints, modules represent models, with generic functions explicitly taking *module-parameters*. [Fig. 14](#) demonstrates some examples. By contrast to concepts and genus constraints, module types and modules do not have type parameters, instead they have type members, such as the  $t$  in the `Eq` module type. `Eq_int` and `Eq_list` are models of `Eq` for the `int` and generic list. Generic functions that need constraints, such as `foo` and `foo'`, explicitly take implicit module parameters `EL` and `E`. Notice that just as type parameters, `EL` and `E` are *compile-time* parameters, not run-time. They are called implicit because at a call to generic function actual models can be inferred, as in the  $x$  and  $y$  examples in [Fig. 14](#). Notice that in the `foo` function any model of comparison of lists is expected, whereas `foo'` expects a

```
module type Eq = sig
  type t
  val equal : t -> t -> bool
end

implicit module Eq_int = struct
  type t = int
  let equal x y = ...
end

implicit module Eq_list {E : Eq} = struct
  type t = Eq.t list
  let equal xs ys = ...
end

let foo {EL : Eq} xs ys = if EL.equal(xs, ys)
  then xs else xs @ ys
let foo' {E : Eq} xs ys = if (Eq_list E).equal(xs, ys)
  then xs else xs @ ys

let x = foo [1;2;3] [4;5]
let y = foo' [1;2;3] [4;5]
```

Fig. 14. OCaml modular implicits

```
concept Equality[T] { bool Equal(T x, T y);
  bool NotEqual(T x, T y) { return !Equal(x, y); }}

concept Ordering[T] refines Equality[T]
{ int Compare(T x, T y); }

interface ISet<T | Equality[T] eq> { ... }
interface ICollection<T> { ...
  bool Remove<Equality[T] eq>(T x); ... }

bool Contains<T | Equality[T] eq>(IEnumerable<T> vs, T x)
{ ... if (eq.Equal(...) ... }

int MaxInt<|Ordering[int] ord>(IEnumerable<int> vs) { ... }
```

Fig. 15. The use of concept-parameters in C#

model of comparison of elements of lists and fixes the model `Eq_list E` of comparison of lists.

2) *Concept Parameters for C#*: [Fig. 15](#) shows some examples of a generic code in the style of concept-parameters, which we call Cp# — C# with concept-Parameters. Concepts are the same as in C#<sup>cp</sup>, whereas constraints on type parameters are not predicates any more, they are explicitly stated as *parameters* in the angle brackets after the “|” sign. In the `ICollection<T>` interface the `Remove` method is obviously generic: it takes the concept-parameter `eq` for comparing the values of the type  $T$ . Note that concept-parameters can even be non-generic as in the `MaxInt` function.

If default models are supported, it must be possible to infer concept-arguments just in the same way as in C# or Genus, so that instances of generic functions and classes can be written in a usual way, without the need to specify the models required:

```
var ints = new ISet<int>(...);
var has5 = Contains(ints, 5);

var maxv = MaxInt(ints);
var minv = MaxInt<|IntOrdDesc>(ints);

ISet<String> s1 = ...;
ISet<String|StringEqCaseIS> s2 = ...;
s1 = s2; // Static ERROR, s1 and s2 have different types
```

C#<sup>cp</sup> and Genus can easily be redesigned to follow the “concept-parameters style” presented here. With this style the syntax of such languages would perfectly fit the semantics. On the other hand, the “concept-predicates style” misleads a

	Haskell	C#	Java 8	Scala	Ceylon	Kotlin	Rust	Swift	JavaGI	G	C# <sup>cp</sup>	Genus	ModImpl
<b>Constraints can be used as types</b>	○	●	●	●	●	●	●	●	●	○	○	○	○
<b>Explicit self types</b>	—	○	○	●	●	○	●	●	●	—	—	—	—
<b>Multi-type constraints</b>	●	*	*	*	○	*	○	○	●	●	●	●	●
<b>Retroactive type extension</b>	—	●	○	○	○	●	●	●	○	○	○	○	—
<b>Retroactive modeling</b>	●	*	*	*	○	*	●	●	●	●	●	●	●
<b>Type conditional models</b>	●	○	○	○	○	○	●	○	●	●	●	●	●
<b>Static methods</b>	● <sup>a</sup>	○	●	○	●	●	●	●	●	● <sup>a</sup>	● <sup>a</sup>	● <sup>a</sup>	● <sup>a</sup>
<b>Default method implementation</b>	●	○	●	●	●	●	●	●	●	●	●	○	○
<b>Associated types</b>	●	○	○	●	○	○	●	●	○	●	●	○	●
<b>Constraints on associated types</b>	●	—	—	●	—	—	●	●	—	●	●	—	●
<b>Same-type constraints</b>	●	—	—	●	—	—	●	●	—	●	●	—	●
<b>Subtype constraints</b>	—	●	●	●	●	●	—	●	○	○	●	○	—
<b>Supertype constraints</b>	—	○	○	●	○	○	—	○	○	○	●	○	—
<b>Concept-based overloading</b>	○	○	○	○	○	○	●	○	○	● <sup>d</sup>	○	○	○
<b>Multiple models</b>	○	*	*	*	*	*	○	○	○	● <sup>b</sup>	●	●	●
<b>Models-consistency (model-dependent types)</b>	— <sup>c</sup>	○	○	○	○	○	— <sup>c</sup>	— <sup>c</sup>	— <sup>c</sup>	— <sup>c</sup>	●	●	●
<b>Model genericity</b>	—	*	*	*	*	*	●	○	○	○	○	●	—

<sup>a</sup>Constraints constructs have no self types, therefore, any function member of a constraint can be treated as static function.

<sup>b</sup>G supports lexically-scoped models but not really multiple models.

<sup>c</sup>If multiple models are not supported, the notion of model-dependent types does not make sense.

<sup>d</sup>C++0x concepts, in contrast to G concepts, provide full support for concept-based overloading.

TABLE I  
THE LEVELS OF SUPPORT FOR GENERIC PROGRAMMING IN OO LANGUAGES

programmer and masks the fact that constraints can be non-uniquely satisfied.

## V. CONCLUSION AND FUTURE WORK

Table I provides a summary on comparison of the languages: each row corresponds to one property important for generic programming; each column shows levels of support of the properties in one language. Black circle ● indicates full support of a property, ○ — partial support, ○ means that a property is not supported at language level, \* means that a property is emulated using the Concept pattern, and the “—” sign indicates that a property is not applicable to a language. The “ModImpl” column corresponds to the OCaml modular implicits. All the properties that appear in rows of Table I were discussed in Sec. III and Sec. IV. Related properties are grouped within horizontal lines; some of them are mutually exclusive. For example, as we saw earlier, using constraints as types and natural language support for multi-type constraints are mutually exclusive properties. The major features analysed in the paper are highlighted in bold.

The purpose of this table is not to determine the best language. The purpose is to show dependencies between different properties and to graphically demonstrate that the “constraints-are-Not-types” approach is more powerful than the “constraints-are-types” one. There are some features that can be expressed under any approach, such as static methods, default method implementations, associated types [15], and even type-conditional models.

It should be mentioned that the table is not exhaustive. There is a bunch of facilities that we did not discuss at all, although they can be considered independently of the study

we made. Thus, for example, Genus [9] provides a support for such useful feature as *multiple dynamic dispatch*. Consider the following code:

```

constraint Intersectable[T] { T T.intersect(T that); }
model ShapeIntersect for Intersectable[Shape]
{ Shape Shape.intersect(Shape s) {...}
  // Rectangle and Circle are subclasses of Shape:
  Rectangle Rectangle.intersect(Rectangle r) {...}
  Shape Circle.intersect(Rectangle r) {...}
  Shape Triangle.intersect(Circle c) {...} ... }

```

It provides a subtype polymorphism on multiple arguments. So that in the call `s1.intersect(s2)` the most specific version of `intersect` would be used depending on the *dynamic* types of `s1` and `s2`.

Another interesting feature is *concept variance*. For example, suppose we have the following Cp# definitions:

```

concept Equality[T] { bool Equal(T x, T y);
  bool NotEqual(T x, T y) { return !Equal(x, y); }}

concept Ordering[T] refines Equality[T]
{ int Compare(T x, T y); }

interface ISet<T | Equality[T] eq> { ... }

```

If `ISet<T|eq>` is covariant on the `eq` in a sense of the refinement relation, then the class `SortedSet<T | Ordering[T] ord>` can legally implement `ISet<T|ord>`. Now recall the `ICollection<T>` interface definition:

```

interface ICollection<T> { ...
  bool Remove<Equality[T] eq>(T x); ... }

```

`SortedSet<T|ord>` obviously also implements the interface `ICollection<T>`. Should it be the case that the `ord` model of `Equality[T]` required in the `Remove` method be used in place of `eq`? Or the `Remove` method has to remain model-generic?

There are other questions similar to mentioned above that relate constraints on type parameters to usual features of object-oriented programming. Some of these questions require a careful type-theoretical investigation, so this is the subject for future work.

#### ACKNOWLEDGMENT

The author would like to thank Artem Pelenitsyn, Jeremy Siek, and Ross Tate for helpful discussions on generic programming.

#### REFERENCES

- [1] Musser D. R. and Stepanov A. A. Generic Programming, *Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, ISAAC '88, London, UK, UK: Springer-Verlag, 1989, pp. 13–25.
- [2] Garcia R. et al. An Extended Comparative Study of Language Support for Generic Programming, *J. Funct. Program.*, Mar. 2007, vol. 17, no. 2, pp. 145–205.
- [3] Bernardy J.-P. et al. A Comparison of C++ Concepts and Haskell Type Classes, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, Victoria, BC, Canada: ACM, 2008, pp. 37–48.
- [4] Garcia R. et al. A Comparative Study of Language Support for Generic Programming, *SIGPLAN Not.*, Oct. 2003, vol. 38, no. 11, pp. 115–134.
- [5] Oliveira B. c. d. s. and Gibbons J. Scala for Generic Programmers: Comparing Haskell and Scala Support for Generic Programming, *J. Funct. Program.*, July 2010, vol. 20, no. 3-4, pp. 303–352.
- [6] Wehr S. and Thiemann P. JavaGI: The Interaction of Type Classes with Interfaces and Inheritance, *ACM Trans. Program. Lang. Syst.*, July 2011, vol. 33, no. 4, 12:1–12:83.
- [7] Siek J. G. and Lumsdaine A. A Language for Generic Programming in the Large, *Sci. Comput. Program.*, May 2011, vol. 76, no. 5, pp. 423–465.
- [8] Belyakova J. and Mikhalkovich S. Pitfalls of C# Generics and Their Solution Using Concepts, *Proceedings of the Institute for System Programming*, June 2015, vol. 27, no. 3, pp. 29–45.
- [9] Zhang Y. et al. Lightweight, Flexible Object-oriented Generics, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, Portland, OR, USA: ACM, 2015, pp. 436–445.
- [10] Martelli A. and Montanari U. An Efficient Unification Algorithm, *ACM Trans. Program. Lang. Syst.*, Apr. 1982, vol. 4, no. 2, pp. 258–282.
- [11] Canning P. et al. F-bounded Polymorphism for Object-oriented Programming, *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, Imperial College, London, United Kingdom: ACM, 1989, pp. 273–280.
- [12] Bruce K. et al. On Binary Methods, *Theor. Pract. Object Syst.*, Dec. 1995, vol. 1, no. 3, pp. 221–242.
- [13] Kennedy A. and Syme D. Design and Implementation of Generics for the .NET Common Language Runtime, *SIGPLAN Not.*, May 2001, vol. 36, no. 5, pp. 1–12.
- [14] Belyakova J. and Mikhalkovich S. A Support for Generic Programming in the Modern Object-Oriented Languages. Part 1. An Analysis of the Problems, *Transactions of Scientific School of I.B. Simonenko. Issue 2*, 2015, no. 2, 63–77 (in Russian).
- [15] Järvi J., Willcock J., and Lumsdaine A. Associated Types and Constraint Propagation for Mainstream Object-oriented Generics, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, San Diego, CA, USA: ACM, 2005, pp. 1–19.
- [16] *The Ceylon Language Specification, version 1.2.2 (March 11, 2016)*.
- [17] *The Kotlin Reference, version 1.0 (February 11, 2016)*.
- [18] *Java Platform, Standard Edition (Java SE) 8*.
- [19] Oliveira B. C., Moors A., and Odersky M. Type Classes As Objects and Implicits, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 341–360.
- [20] Pelenitsyn A. Associated Types and Constraint Propagation for Generic Programming in Scala, English, *Programming and Computer Software*, 2015, vol. 41, no. 4, pp. 224–230.
- [21] *The Rust Reference, version 1.7.0 (March 3, 2016)*.
- [22] Hall C. V. et al. Type Classes in Haskell, *ACM Trans. Program. Lang. Syst.*, Mar. 1996, vol. 18, no. 2, pp. 109–138.
- [23] Wadler P. and Blott S. How to Make Ad-hoc Polymorphism Less Ad Hoc, *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, Austin, Texas, USA: ACM, 1989, pp. 60–76.
- [24] Stroustrup B. *Concept Checking — A More Abstract Complement to Type Checking*, Technical Report N1510=03-0093, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, Oct. 2003.
- [25] Stroustrup B. and Dos Reis G. *Concepts — Design Choices for Template Argument Checking*, Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, Oct. 2003.
- [26] Dos Reis G. and Stroustrup B. Specifying C++ Concepts, *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, Charleston, South Carolina, USA: ACM, 2006, pp. 295–308.
- [27] Stroustrup B. and Sutton A. *A Concept Design for the STL*, Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, Jan. 2012.
- [28] Stepanov A. A. and Lee M. *The Standard Template Library*, Technical Report 95-11(R.1), HP Laboratories, Nov. 1995.
- [29] Sutton A. *C++ Extensions for Concepts PDTS*, Technical Specification N4377, ISO/IEC JTC1/SC22/WG21, C++ Standards Committee Papers, Feb. 2015.
- [30] Greenman B., Muehlboeck F., and Tate R. Getting F-bounded Polymorphism into Shape, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 89–99.
- [31] White L., Bour F., and Yallop J. Modular Implicits, *ArXiv e-prints*, Dec. 2015, arXiv: 1512.01895 [cs.PL].

# Refinement Types in Jolie

Alexander Tchitchigin<sup>\*†</sup>, Larisa Safina<sup>\*</sup>, Mohamed Elwakil<sup>\*‡</sup>, Manuel Mazzara<sup>\*</sup>, Fabrizio Montesi<sup>§</sup> and Victor Rivera<sup>\*</sup>

<sup>\*</sup>Innopolis university, Innopolis, Russia

Email: {a.chichigin, l.safina, m.elwakil, m.mazzara, v.rivera}@innopolis.ru

<sup>†</sup>Kazan Federal University, Russia. Email: a.tchichigin@it.kfu.ru

<sup>‡</sup>Cairo University, Giza, Egypt. Email: m.elwakil@fci-cu.edu.eg

<sup>§</sup>University of Southern Denmark, Denmark

Email: fmontesi@imada.sdu.dk

**Abstract**—Jolie is the first language for microservices and it is currently dynamically type checked. This paper considers the opportunity to integrate dynamic and static type checking with the introduction of refinement types, verified via an SMT solver. The integration of the two aspects allows a scenario where the static verification of *internal* services and the dynamic verification of (potentially malicious) *external* services cooperate in order to reduce testing effort and enhance security.

**Index Terms**—Microservices, Jolie, Refinement Types, SMT, SAT, Z3

## I. INTRODUCTION

“Stringly typed” is a new antipattern referring to an implementation that needlessly relies on strings, when other options are available. The problem of “string typing” appears often in service-oriented architecture and microservices on the border between a service and its clients (external interfaces) due to necessity to communicate over text-based protocols (like HTTP) and collaboration with clients written in dynamically-typed languages (like JavaScript). The solution to this problem can be found with refinement types [14], which are used to statically (or dynamically) check compatibility of a given value and refined type by means of predicates constraining the set of possible values. Though employment of numerical refinements is well-known in programming languages, string refinements are still rare.

In this paper, we introduce a design for extending the Jolie programming language [25], [3] and its type system. On top of previous extensions with choice type [28] and regular expressions, we introduce here string refinement type and we motivate the reasons for such extension. Section II recalls the basic of the Jolie language and its type system while Section III describes the open problem this paper tackles with clarifying examples. Section IV discusses related work in the context of using SMT solvers for static typing of refinement types.

## II. JOLIE PROGRAMMING LANGUAGE

Jolie [25] is the first programming language based on the paradigm of microservices [18]: all components are autonomous services that can be deployed independently and operate by running parallel processes, programmed following the workflow approach. The language was originally developed in the context of a major formalization effort for

workflow and services composition languages, the EU Project SENSORIA [1], which spawned many models for reasoning on the composition of services (e.g., [19], [20]). Jolie comes with a formally-specified semantics [16], [15], [24], inspired by process calculi such as CCS and the  $\pi$ -calculus [21]. On the more practical side, Jolie is inspired by standards for Service-Oriented Computing such as WS-BPEL [4]. The combination of theoretical and practical aspects in Jolie has made it a candidate for the application of recent research methodologies, e.g., for addressing runtime adaptation [27], process-aware web applications [23], or correctness-by-construction in concurrent software [9].

Jolie has the microservice as basic abstraction and as first-class citizen, and it is based on a recursive model where every microservice can be easily reused and composed for obtaining, in turn, other microservices [22]. This approach supports distributed architecture and guarantees simple managing of components, which reduces maintenance and development costs.

Microservices work together by exchanging messages. In Jolie, messages are structured as trees [24] (a variant of the structures that can be found in XML or JSON). Communications are type checked at runtime, when messages are sent or received. Type checking of incoming messages is especially relevant, since it mitigates the effect of ill-behaved clients. The work of Nielsen [26] presents a first attempt at formalizing a static type checker for the core fragment of Jolie. However, for the time being, the language is still dynamically type checked.

## III. EXTENSION OF JOLIE TYPE SYSTEM

Safina et al [28] extended the basic type system of Jolie with type choices. The work had been then continued with the addition of regular expression types, a special case of refinement types. In refinement types, types are decorated with logical predicates which further constrain the set of values described by the type and therefore represent the specification of invariant on values. Here, we extend this with the possibility of expressing invariants on string values in form of regular expressions.

The integration of static and dynamic analysis allows considering “*internal*” services (native Jolie services) and calls from “*external*” services (potentially developed in other



languages) in a complementary way. The first ones can be statically checked while the second ones, which could exhibit malicious behavior, still need a runtime validation.

The key idea behind service-oriented computing, and microservices in particular, is the ability to connect services developed in different programming languages and possibly running on different servers over standard communication protocols [18]. A common use case is the implementation of APIs for Web and mobile applications. In such scenarios, the de-facto standard communication protocol is HTTP(S), combined with standardized data formats (SOAP, JSON, etc.).

HTTP is a text-based protocol, where all data get serialized into strings<sup>1</sup>. Moreover, clients of a service (an application or another service) may have been developed in a language that does not support particular datatypes (e.g., JavaScript does not have a datatype for calendar dates or time of day), therefore relying on string representation for internal processing too. The same issue arises with key-value storage systems (e.g., Memcache and Redis), which support only string keys and string values. These factors make string handling an important part of a service application, especially at the boundary with external systems.

Not all strings are made equal. For example, GUIDs are often used to identify records in a store. GUIDs are represented as strings of hexadecimal digits with a particular structure. Currently, developers have to manually check the conformance of received values to the expected format. In such a scenario, a developer has to find her way in a narrow stream between the *Scylla* of forgetting to insert necessary checks and the *Charybdis* of inserting too many checks for data that has been already validated<sup>2</sup>.

Description of the *shape* of expected string data (like GUID or e-mail address) is natural with regular expressions. Adding the description of this *shape* to the datatype definition allows the compiler to automatically insert the necessary dynamic checks (for public functions) and statically validate the conformance (for internal calls). This is the extension of refinement type to string type. The same techniques and tools used for static verification of conformance for numerical refinements [17], [12] can be used for strings. For the purposes of this paper we will use Z3 SMT solver by Microsoft Research [6], which recently got support for theory of strings and regular expressions in its development branch.

#### A. Example: the news board

The approach to static checking of string refinements using Z3 SMT solver is illustrated here by a simple example, i.e. a service using refined datatype for GUIDs and the SMT constraints generated for it.

<sup>1</sup>Jolie partially mitigates this aspect with automatic conversion of string serializations to structured data by following the interface definition of the service [23]. However, this does not solve the general problem addressed here.

<sup>2</sup>Scylla and Charybdis are monsters of Greek mythology living on the two sides of a narrow channel so that sailors trying to avoid one would have fallen into the other.

A *news board* is a simple service in charge of retrieving posts composed by a particular user of the system. The service receives user information via HTTP in a string format. Here we use string refinement types to define the shape of user IDs (employing regular expression that matches GUIDs) as an alternative to the manual checking of the constraint inside the posts retrieving operation.

---

```
1 type guid: string ("[A-F\\d]{8,8}-[A-F\\d]{4,4}-[A-F\\d]{4,4}-[A-F\\d]{4,4}-[A-F\\d]{12,12}")
```

---

Types for storing user and posts information are also necessary<sup>3</sup>.

---

```
1 type user: void {
2   .uid: guid
3   .name: string
4   .age: int (age > 18) }
5 type post_type: void {
6   .pid: guid
7   .owner: guid
8   .content: string }
9 type posts: void { .post*: post_type }
```

---

We leave service deployment information out of this paper due to its low relevance to the topic, the full code example can be found in [2]. The behavioral fragment of the *news board* demonstrates the post retrieval for a particular user. To get the information the right user has to be found (*find\_user\_by\_name*) and pass the GUID to *get\_all\_users\_posts*.

There are two definitions of the operation in the following code fragment: *all\_posts\_by\_user* and *all\_posts\_by\_user2*. In the first one the correct data is passed to *get\_all\_users\_posts*, i.e. *user.uid*; while in the second *user.name* is passed. Without string refinement a problem would arise. The code is syntactically correct. However, it's semantically incorrect since no information can be retrieved by user's name when user's ID is actually expected.

---

```
1 main {
2   all_posts_by_user (name) {
3     find_user_by_name@SelfOut(name)(
4       user);
5     get_all_users_posts@SelfOut(user.
6       uid)(posts) };
7   all_posts_by_user2 (name) {
8     find_user_by_name@SelfOut(name)(
9       user);
10    //and here we pass the wrong field!
11    get_all_users_posts@SelfOut(user.
12      name)(posts) };
```

---

<sup>3</sup>Please note that in Jolie we structure the variable's data as a tree, where the nodes contain values. Using the *void* type for the variable on the top of the tree, we show that it contains no data and is used as a container for its subtrees.

```

10
11 //find_user_by_name definition
12 //get_all_users_posts definition
13 }

Introducing string refinement allows Jolie to have both
dynamic and static checking for strings. In case of dynamic
checking, the string is verified at runtime when passed to the
receiving service. The more interesting case is static checking
by means of SMT. Here we present the most essential parts
of the encoding, complete example can be found in [2].

1; notions of types, terms and typing
   relation
2(declare-sort Type)
3(declare-sort Term)
4(declare-fun HasType (Term Type) Bool)
5
6; type of strings of a programming
   language
7(declare-fun string () Type)
8; translation from Z3 built-in String
   type to our string type and back
9(declare-fun BoxString (String) Term)
10(declare-fun string-term-val (Term)
    String)
11(assert (forall ((str String))
12  (= (string-term-val (BoxString str))
    str)))
13(assert (forall ((s String))
14  (HasType (BoxString s) string)))
15
16; guid type that refines string type
17(declare-fun guid () Type)
18(define-fun guid-re () (RegEx String))
19; the construction of the regular
   expression is omitted
20)
21; refinement definition for guid type
22(assert (forall ((x Term))
23  (iff (HasType x guid)
24    (and (HasType x string)
25      (str.in.re (string-term-val x)
        guid-re)))))
26; we define type 'user' through it's
   projections
27(declare-fun user () Type)
28(declare-fun user.uid (Term) Term)
29(declare-fun user.name (Term) Term)
30(declare-fun user.age (Term) Term)
31; typing rules for projections
32(assert (forall ((t Term))
33  (implies (HasType t user)
34    (and (HasType (user.uid t) guid)
35      (HasType (user.name t) string)
36      (HasType (user.age t) nat)))))
37

```

```

38(declare-fun find_user_by_name (Term)
   Term)
39; find_user_by_name : string -> user
40(assert (forall ((name Term))
41  (implies (HasType name string)
42    (HasType (find_user_by_name name)
      user))))
43
44; type checking for all_posts_by_user
45(assert (not (forall ((t Term))
46  (implies (HasType t string)
47    (HasType (user.uid (
48      find_user_by_name t)) guid)))))
49; type checking for all_posts_by_user2
49(assert (not (forall ((t Term))
50  (implies (HasType t string)
51    (HasType (user.name (
52      find_user_by_name t)) guid)))))

```

Type checking is based on proving a theorem stating that a function is correctly typed. Technically, the opposite proposition is actually stated and the SMT solver is put in charge of finding a counterexample. A failure in such an attempt leads to the conclusion that the original theorem has to be true (proof by contradiction).

The Z3 solver successfully proves the well-typedness theorem for the correct implementation of *all\_posts\_by\_user*, and fails to disprove the incorrect implementation (*all\_posts\_by\_user2*) due to many simplifications to the presented SMT encoding for the sake of clarity and understandability. Employment of a more sophisticated encoding for the actual implementation of refinement constraints may mitigate this situation and is left as future work.

#### IV. RELATED WORK

Within the context of functional languages, type-checking of refined types by employing SMT solvers is not new. In [7], the authors present the design and implementation of the F7 enhanced type-checker for the functional language F# that verifies security properties of cryptographic protocols and access control mechanisms using Z3 [10]. The SAGE language [17] employs a hybrid approach [13] that performs both static and dynamic type-checking. During compilation time, the Simplify theorem prover [11] is used to check refinement types. If Simplify is not able to decide a particular subtyping relation, a proper type cast is inserted in the code and it is checked at runtime. If the type cast fails during runtime, this particular subtyping relation is inserted in a database of known failed casts. In contrast to checking syntactic subtyping as in F7 and SAGE, the authors of [8], introduce semantic subtyping checking for a subset of the M language [5] using the Z3 SMT solver.

#### V. CONCLUSIONS

The Jolie language is dynamically type-checked. This paper explores the possibility of integrated dynamic and static type

checking with the introduction of refinement types, verified via an SMT solver. The integration of the two aspects allows a scenario where the static verification of *internal* services and the dynamic verification of (potentially malicious) *external* services cooperates in order to reduce testing effort and enhance security.

In this work, we motivate the usefulness and feasibility of string refinement types using an example. Naturally we need to integrate this extension with an actual type-checker employing a more advanced SMT-encoding. Not only for strings but for numerical types too which is well-known and useful tool for correctness enhancement.

When we have a type-checker for refinement types, an interesting empirical study would be checking of existing programs augmented with refined types to discover whether this technique can uncover bugs caused by a developer's oversight.

## REFERENCES

- [1] EU Project SENSORIA. Accessed April 2016. <http://www.sensoria-ist.eu/>.
- [2] Gist of SMT constraints for the example. Accessed April 2016. <https://gist.github.com/gabriel-fallen/a04c33860e2157201fa8>.
- [3] Jolie Programming Language. Accessed April 2016. <http://www.jolie-lang.org/>.
- [4] WS-BPEL OASIS Web Services Business Process Execution Language. accessed April 2016. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html>.
- [5] Power Query formula reference. Technical Report, August 2015.
- [6] Microsoft Research. Accessed April 2016. Z3. <https://github.com/Z3Prover/z3>.
- [7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, February 2011.
- [8] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hrițcu, and David Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10*, pages 105–116, New York, NY, USA, 2010. ACM.
- [9] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
- [10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- [11] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, May 2005.
- [12] Joshua Dunfield. *A unified system of type refinements*. PhD thesis, Air Force Research Laboratory, 2007.
- [13] Cormac Flanagan. Hybrid type checking. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '06*, pages 245–256, New York, NY, USA, 2006. ACM.
- [14] Tim Freeman and Frank Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, May 1991.
- [15] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.
- [16] Claudio Guidi, Roberto Lucchi, Gianluigi Zavattaro, Nadia Busi, and Roberto Gorrieri. Sock: a calculus for service oriented computing. In *ICSOC, volume 4294 of LNCS*, pages 327–338. Springer, 2006.
- [17] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report), 2006.
- [18] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. Accessed April 2016. <http://martinfowler.com/articles/microservices.htm>.
- [19] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
- [20] Manuel Mazzara, Faisal Abouzaid, Nicola Dragoni, and Anirban Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations - A process algebra perspective. In *Web Services and Formal Methods - 8th International Workshop, WS-FM*, pages 64–78, 2011.
- [21] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.
- [22] Fabrizio Montesi. JOLIE: a Service-oriented Programming Language. Master's thesis, University of Bologna, 2010.
- [23] Fabrizio Montesi. Process-aware web programming with Jolie. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*, pages 761–763, New York, NY, USA, 2013. ACM.
- [24] Fabrizio Montesi and Marco Carbone. Programming Services with Correlation Sets. In *Proc. of Service-Oriented Computing - 9th International Conference, ICSOC*, pages 125–141, 2011.
- [25] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.
- [26] J. M. Nielsen. A Type System for the Jolie Language. Master's thesis, Technical University of Denmark, 2013.
- [27] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbriellini. AIOCJ: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, pages 161–170, 2014.
- [28] Larisa Safina, Manuel Mazzara, Fabrizio Montesi, and Victor Rivera. Data-driven workflows for microservices (genericity in jolie). In *Proc. of The 30th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2016*.

# Visual Dataflow Language for Educational Robots Programming

Grogorii Zimin  
*Mathematics and Mechanics Faculty,  
SPbSU  
Saint-Petersburg, Russia  
Email: zimin.grigory@gmail.com*

Dmitrii Mordvinov  
*Mathematics and Mechanics Faculty,  
SPbSU  
Saint-Petersburg, Russia  
Email: mordvinov.dmitry@gmail.com*

**Abstract**—The paper describes a novel dataflow visual programming environment for embedded robotic platforms. Its purpose is to be "bridge" between lightweight educational robotic programming languages and complex industrial languages. We compare programming environments mostly used by robotics community with our tool. A brief review of behavioural robotic architectures and some thoughts on expressing them in terms of our language are given. We also provide the examples of solving two typical robot control tasks in our language.

## 1. Introduction

Programming languages for creating robotic controllers are actual topics of research oftenly discussed at major conferences, such as ICRA [1] or IROS [2]. Visual programming languages (VPLs) are also actively discussed for the last three decades, the largest conferences are held annually, e.g. VL/HCC [3]. VPLs are oftenly applied in robotics domain [4–8] allowing to create and visualize robotic controllers. Robotic VPLs are commonly used for educational purposes, making possible for students of even junior schools to create robotic programs. For these aims there are already exists a great number of educational robotic programming environments based on VPLs, e.g. NXT-G [9], TRIK Studio [10], ROBOLAB [11], also there are some academic tools implementing interesting and novel approaches to educational robotics programming [4], [6], [8].

Robotic control programs are inherently reactive: they transform data which is continuously coming from multiple sensors into the impulses on actuators. For this reason dataflow languages (DFLs) are well-suitable for robotics programming. Many researchers denoted the conveniency of dataflow visual programming languages (DFVPLs) [12], finding them more useful than textual DFLs, for example because data flows explicitly displayed on the diagram. There are large and complex general-purpose and domain-specific development environments such as LabVIEW [13] and Simulink [14] that provide a large (and sometimes even cumbersome) set of libraries for robotics programming. More detailed discussion of robotics VPLs will be provided in section 2.

There is a large number of robotic constructor kits for learning the basics of robotics and cybernetics, such as LEGO MINDSTORMS [15], TRIK, ScratchDuino [16]. Modern programming languages which are used for programming those kits are based on the control flow model rather than on dataflow model. Control flow-based languages are good for solving scholar "toy" tasks, but may be inconvenient for programming more complex "real world" controllers that may be conveniently expresses on DFLs. The simple DFVPL may be considered as a useful step from educational VPLs to the programming languages which are used in universities and industry.

This paper discusses a novel extensible tool for programming all popular educational robotic kits on dataflow visual programming language. It should be noted that, in distinction from other tools, our tool is focused on embedded systems (section 6). Another interesting detail of our work is the application of DSM-approach for implementation of visual editor: it is entirely generated by QReal DSM-platform [17] [18] without even a line of code written. We also take into consideration the popularity of Brooks' Subsumption Architecture [19] which is still mainstream approach to design of complex robotic controllers [4], [5], [7], [20] despite it was proposed 30 years ago. Brooks' Subsumption Architecture and some other are conveniently expressed in our language, they are discussed in section 3.

The remainder of a paper is organized as follows. An overview of robotics VPLs and DFVPLs is presented in section 2. Section 3 provides some general thoughts on how some widely used robotic behavioural architectures are expressed in our language. A detailed description of our language is given in section 4. Section 5 demonstrates two typical robotic controllers expressed in our language. The most important details of implementation are discussed in section 6. Finally, the last section concludes the paper and discusses possible directions for future work.

## 2. Similar Tools

Robot programming environments can be divided into three categories: educational, which allows to program small educational robotic kits; industrial, which have a rich toolkit for creating large and complex robotic controllers; academic,

which implement new interesting ideas, however they are oftenly unavailable for downloading or unusable.

Educational visual environments are for example NXT-G and ROBOLAB for LEGO MINDSTORMS NXT kit, EV3 Software for the Lego Mindstorms EV3 kit, TRIK Studio for NXT, EV3 and TRIK. Those environments simplify solving primitive robot control tasks like finding a way out of the maze and driving along the line using light sensors, which makes the process of learning the basics of programming and robot control easy. But their simplicity oftenly bounds the flexibility of the language. Visual languages of all mentioned systems are based on control flow model.

There is also a number of well-known visual robotic programming environments of industrial level. For example, general-purpose LabVIEW from National Instruments with the DfVPL G, programming environment Simulink developed by MathWorks for modelling different dynamic models or control systems. Those products offer a huge set of models and libraries to create control systems, test benches, real-time systems of any complexity, using model-driven approach. LabVIEW provides opportunity for programming small robots. There are lots of examples of applying LabVIEW in education [21], [22], but much more oftenly adaptations like Robolab are used in educational process. It should be noted that those environments are distributed under the commercial license.

Another example of an visual robotics industrial system is the Microsoft Robotics Developer Studio (MSRDS) [23], which is free for academic purposes and allow to create distributed robotic systems on DfVPL. MSRDS officially supports a large set of robotic platforms, LEGO NXT [24] in particular (however, the autonomous mode for NXT is not supported). MSRDS has the ability of manual integration with custom robotic platforms, but unhappily is not maintained since 2014.

There is a lot of scientific research has done in this area, e.g., dissertation [4] describes a visual programming module for expressing robotic controllers in terms of extended Moore machines, [6], [7] describe visual environment for *occam- $\pi$*  language and *Transterpreter* framework, and its usage in education and swarm robotics. Article [8] describes DfVPL for beginners which is pretty close to a one we introduce here. However at the moment RuRu is under development, it has pretty limited functionality and even unavailable for download.

### 3. Robotic Behavioural Architectures

The task of creation complex and scalable robotic controller is indeed a non-trivial task. Starting from middle 80's many researchers have attempted to solve this problem and a number of behavioural robotic architectures were proposed [25]. Those approaches are quickly became popular in robotics community and they are still actual. For example the original work that introduced Brooks' *Subsumption Architecture* [19] is one of the most cited works in the entire robotics domain. We believe that the description of modern

language for programming robotic controllers should contain at least general thoughts on how those architectures may be expressed in it.

A controller built on Brooks' Subsumption Architecture is decomposed into a hierarchy of levels of competence where each new layer describes a new feature of robot's behaviour. Levels are "ordered" upside-down, the higher levels describe more "intelligent" behaviour of robot. Higher levels depend on lower ones but not vice versa, so failures of higher levels do not imply the failure of lower. This is important feature for mobile robotics, e.g. if robot's gripper was damaged the controller is still able to deliver robot to its base. Levels of responsibility are expressed as a set of "behaviours" running concurrently and interacting with each other via channels of *suppression* and *inhibition*. Using them higher levels can suppress the activity of lower ones thus correcting the behaviour of the whole system.

Brooks' in his original work offered to express behaviours in terms of *state machines*. Each layer implements some simple logic of transformation sensor inputs into impulses on actuators. Dataflow languages are obviously as suitable as state machines for expressing such behaviours. In our language each behaviour can be represented as "black box" described by separate subprogram. Also our language contains *Suppressor* and *Inhibitor* elements for layers communication. Levels can be invoked concurrently, so we can conclude that our language allows the convenient expression of controllers built with Subsumption Architecture. That is demonstrated by an example in section 5.

Connell's *Colony Architecture* [26] is a very similar to Brooks' one, but solves some scalability issues of Subsumption Architecture. It also decomposes the controller into a number of communicating concurrent levels, but they are unordered. The other difference is an absence of inhibition channel, data inhibition should be implicitly expressed by predicated in layers. Our language does not force any order between layers, predicative inhibition can be implemented simply with *Filter* block. So Colony Architecture is also well-expressed in our language.

There also exist Arkins *Motor Schema* [27] and Rosenblatts *Distributed Architecture for Mobile Navigation (DAMN)* [28] which are compatible with our language, but the detailed descriptions will be omitted here. General ideas on their implementation on *occam- $\pi$*  language can be found in [25], we believe that those ideas will suffice in the context of this paper. The complete research of expressing behavioural architectures in our language is a topic for separate paper.

### 4. Language Description

Evolution of a domain-specific modeling (DSM) tools allows to quickly create a fairly sophisticated visual programming languages [29]. TRIK Studio programming environment is an example of a system that was created using DSM-based approach on QReal platform [17], [18]. Basing on an industrial experience of TRIK Studio developers we

decided to create the visual editor of our language on QReal platform.

Program on DFVPL is a set of blocks and flows that connect blocks. DFVPL blocks process incoming tokens and emit resulting data into the output data flows. Blocks in our language can be divided into several groups that are described below. Some blocks require to specify information on textual language. The language we use is a statically typed dialect of Lua [30].

- *Control* blocks that implement basic algorithmic constructions (conditions, loops, etc).
  - *ConstValue* and *RandomValue* blocks that are responsible for generation of a random number or a predetermined value of any type.
  - *Loop*, *If*, *Switch*. These blocks implement general control flow algorithmic constructions in dataflow style. *Loop* is an entity which emits a sequence of numbers for a given amount of times. *If* checks the condition specified on a textual language and sends them to *True* or *False* channel. *Switch* successively checks guard conditions and if it is evaluated as *true* sends incoming data to corresponding channel.
  - *Function* block, which allows to process of the input data in a textual language. Most usually this block is used for mathematical processing of data.
  - *FinalBlock* stops the execution of program when receiving any data.
  - *Subprogram* for reusing the code. Double-click on subprogram block opens new visual editor tab with an implementation of this subprogram. Contents of that tab can be then edited by user in exactly the same way he edits the main diagram.
  - *GetSetVariable*. Purely practical block for setting value of some global variable or emitting it into output flows.
  - *Wait* block delays data processing.
  - *DelayAndFilter* is the extension of the previous block adding the filtering condition and checking the amount of emitted data validated by condition.
  - *Fork*, *EndFork* blocks that provide an ability of invoking code in platform-specific execution units. See section 6 for details.
- *Drawing*. Blocks for drawing on display of the robot and on the floor in simulator mode.
  - *PaintSettings* defines current background color, thickness and color of pen and color and style of the brush that draw graphical primitives.
  - *ShapePainter*, *SmilePainter*, *Text* are used for drawing some shape, text or smile on robot's display.
  - *Clear* block removes all graphics from robot's display when receiving any token.
  - *Pen* block puts down or raises the marker for drawing the robot's trace on the "floor" of 2D simulator.
- *Flow manipulation*. These elements provide opportunity to manipulate data which flow between blocks.
  - *InPort*, *OutPort* emit tokens that come into some instance of *Subprogram* block into a diagram implementing it and similarly redirect data from subprogram

implementation into output flows of active instance of *Subprogram* block.

- *Suppressor*, *Inhibitor* inhibit or replace token of some flow with tokens of another. These, *Subprogram* and *Fork* blocks provide a compatibility with the Brooks' Subsumption Architecture.
- *Zip*, *Unzip* provide an opportunity to gather data from several *Flows* into one and vice versa.
- *Actions* provide an ability to query and modify state of robot's input and output devices.
  - *Sensor* continuously emits data from specified sensor, e.g. infrared, light, etc.
  - *Servo*, *Motors* process received data and send impulses to robot actuators.
  - *Encoders* block sets the motors tacho limit when receiving data and continuously emits encoder values into output flows.
  - *SendMessage*, *ReceiveMessage* responsible for the coordination of a group of robots.
  - *Say*, *PlayTone*, *LED* responsible for managing speakers and LED lights.
  - *RemoveFile*, *WriteToFile*, *ReadFile* implement working with file system.
  - *InitCamera*, *DetectByVideo*, *StreamingNode* wrap some algorithms of computer vision.
  - *PortBlock* provides an ability to write low-level to some port of the robot.
  - *SystemCall* responsible for the command execution by command line interpreter, e.g. token "reboot" will reboot robot.
  - *Gamepad* reads data from the operator's control device, e.g. gamepad, and emits it.

These blocks are enough to express a pretty wide range of the robotic controllers of varying complexity. If several blocks emitting data from one input device are met only one of them is active. That detail distinguishes our tool from other implementing data flow paradigm, for details see section 6. For example figure 1 shows diagram with *Motors*, *ConstValue*, *Encoders*, *Flows* where *Encoders* block is presented twice. When interpretation started *ConstValue* emits data to *Motors* and *Encoders* (a) emits a value of a tacho counter. When block *Encoders* (b) receives some data and thus nullifies encoder value, at that moment *Encoders* (a) stops emitting tokens.

One important detail about our language is that it explicitly supports control flow model, that is important for educational goals. On figure 1 *ConstValue* and *Motors* have incoming and outgoing "arrows", which are used to connect control flow data. For example *Motors* block emits data to control flow channel when handle incoming data and *ConstValue* emits its value when receives control flow token.

Flows may be pinned to a block on left, right and bottom side, which are highlighted when user edits block (see Figure 2). Also block may contain text fields, e.g. on Figure 2 user entered textual condition.

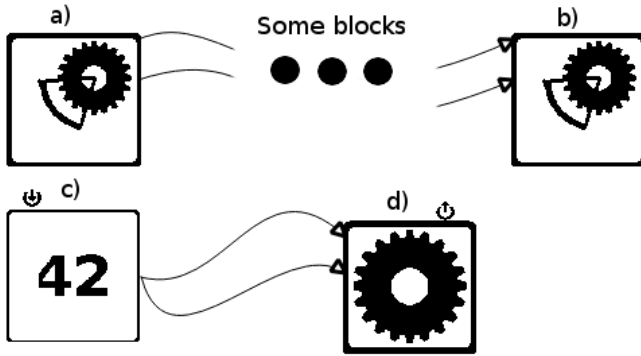


Figure 1: Block with many representations but only one of them can be active. a,b — *Encoders* c — *ConstValue* d — *Motors*

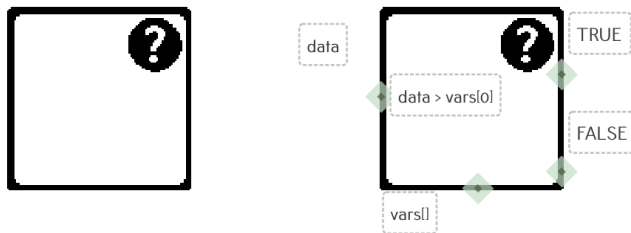


Figure 2: Showing and editing of block.

## 5. Example

Figures 3, 4 show simple PD-regulator which keeps robot on a certain distance from a wall using infrared sensor. Global variable is used for storing old sensor values. Expressions in *Function* block are calculated in upside-down order, results of previous expressions are available on lower levels. Each level emits resulting token into a corresponding flow, in our example two flows are connected directly to motors control block.

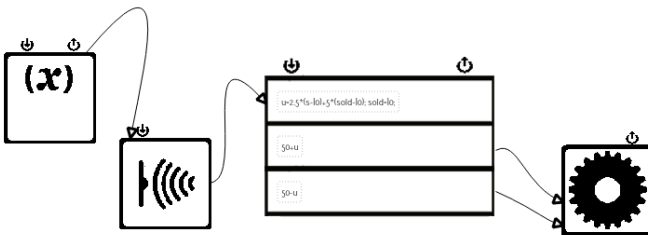


Figure 3: Controller for the wall following.

Let's describe more complex robotic controller. We have the robot equipped with two power motors and two frontal infrared sensors positioned at an angle of 30 degrees on either side of the longitudinal line of symmetry of the robot. Let's consider the robot control system that manages robot

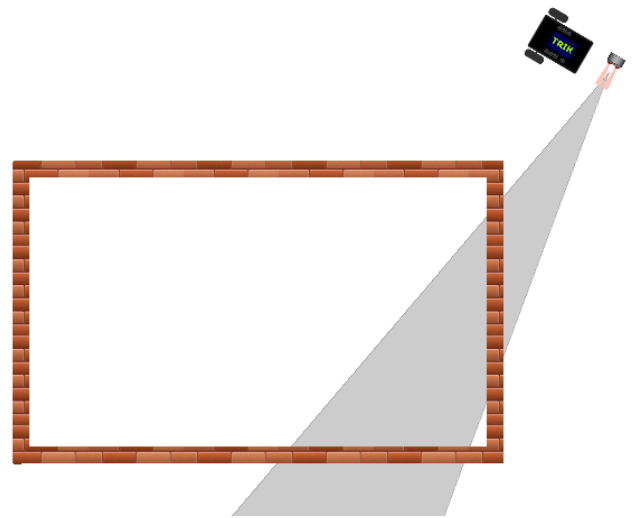


Figure 4: Simulation process of the wall following.

wandering in space and avoiding frontal collisions. But at the same time it allows manual control with gamepad. We divide the problem into three levels responsibility using Subsumption Architecture. The first will be responsible for aimless movement of the robot. The second is responsible for collision avoidance: if the robot is too close to a collision, it must avoid the obstacles preventing robot wandering. The third will be responsible for maintenance of the user queries, the user obtains a full control, the previous levels are suppressed.

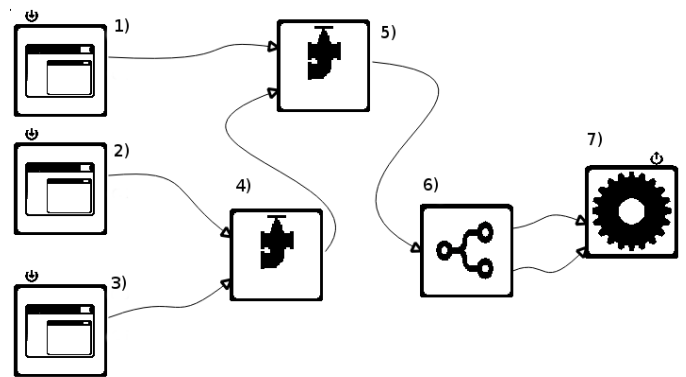


Figure 5: Controller code with three competencies level. 1 — Human control level. 2 — Collision avoidance level. 3 — Wandering level. 4 — *Suppressor* block for levels 2,3. 5 — *Suppressor* block for levels 1 and 2,3. 6 — *Unzip* block. 7 — *Motors* block.

Figure 5 shows this decomposition. Each level represented as *Subprogram* and emits pulses to actuators. Execution begins with the launch of all levels concurrently. Robot wanders aimlessly. If the robot is close to the collision, the Collision avoidance level suppresses the flow with data emitted by Wandering level. If the user starts to



manipulate with the gamepad, the data sent suppress levels described above.

Each level is the simple robot controller without direct connection to actuators. Wandering (first level) continuously generates random number for each robot actuator, and sends its outside as array (see Figure 6). The execution of this level starts with *InPort* which emits data to activate two *RandomValue* blocks. Each *RandomValue* generate random number and emits it to *Wait* block which after some predefined delay sends it to *Zip* block which produces an array storing output values.

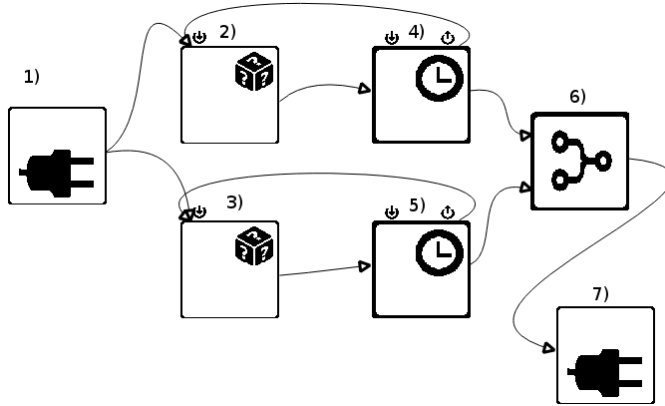


Figure 6: Walking. 1 — *InPort* block. 2,3 — *RandomValue* blocks. 4,5 — *Wait* blocks. 6 — *Zip* block. 7 — *OutPort* block.

The second level is needed to prevent collisions (see Figure 7). It continuously gathers data by *Zip* from two infrared *Sensors* and checks if collision threatens (continuously after some delay by *DelayAndFilter*). If the collision can occur values sent for actuators to evade obstacles are calculated by *Function*. *Function* block emits it to *Zip* block which produces an array storing output values.

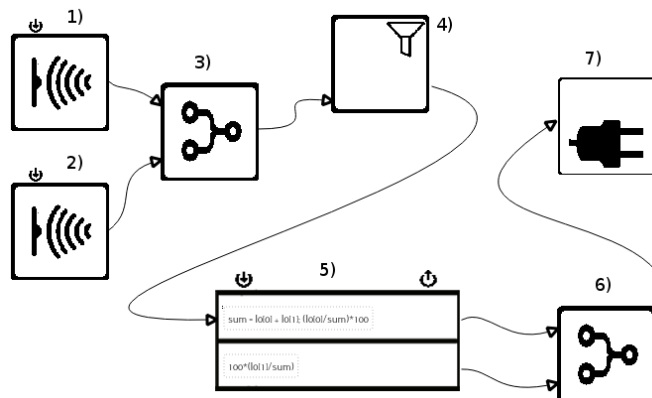


Figure 7: Collision avoidance. 1,2 — *Sensor* blocks. 3,6 — *Zip* block. 4 — *DelayAndFilter* block. 5 — *Function* block. 7 — *OutPort* block.

The third level is responsible for gamepad control. *Gamepad* emits tokens describing current joystick and but-

tons state. For simplicity we assume that pressing any button on gamepad will terminate the robot control program (by *FinalBlock*). The tokens are converted from the *Gamepad* to the array of pulses for actuators by *Function* block, which emits it through *OutPort* block.

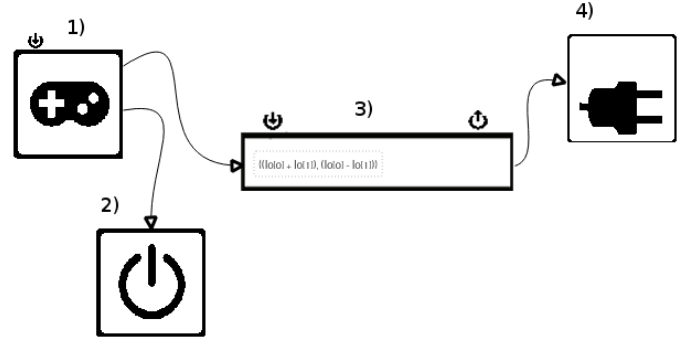


Figure 8: Human control. 1 — *Gamepad* block. 2 — *FinalBlock*. 3 — *Function* block. 4 — *OutPort* block.

## 6. Implementation

The system is implemented as two plugins for TRIK Studio. The first one describes the visual language and provides visual editor for our system. It contains the meta-model of dataflow visual language and entirely generated by QReal DSM-platform. Plugged into TRIK Studio this module provides fully operational visual editor with all advantages of TRIK Studio control flow editor like modern-looking user interface, ability to create elements with mouse gestures, different appearances of links and so on. The time spent on the development of this plugin (not considering discussing and designing the prototype of visual language on paper) roughly equals three man-days. The benefit on exploiting the DSM-approach is obvious, the development of the similar editor from scratch would have been taken vastly more time.

The second plugin contains implementation of dataflow diagrams interpreter. Given the program drawn in editor (provided by first plugin) Interpreter will transform given program which is drawn in editor (provided by first plugin) into a sequence of the commands sent to a target robot (see fig. 9). The target robot can be one of the supported in TRIK Studio infrastructure: Lego NXT or EV3 robot, TRIK robot, TRIK Studio 2D simulator or V-REP 3D simulator [31]. Commands are sent via high-level TRIK Studio devices API, a part of it presented at fig. 10.

The general architecture of interpreter plugin is presented at fig. 11. Given dataflow diagram interpreter traverses, validates and prepares it for interpretation process. For each visited dataflow block implementation object is instantiated. Implementation objects are written in C++. Instantiation is performed by corresponding factory object. Implementation objects are then subscribed each to other like they are connected by flows on diagram, *publish-subscribe* pattern is used here. The set of initial blocks is determined



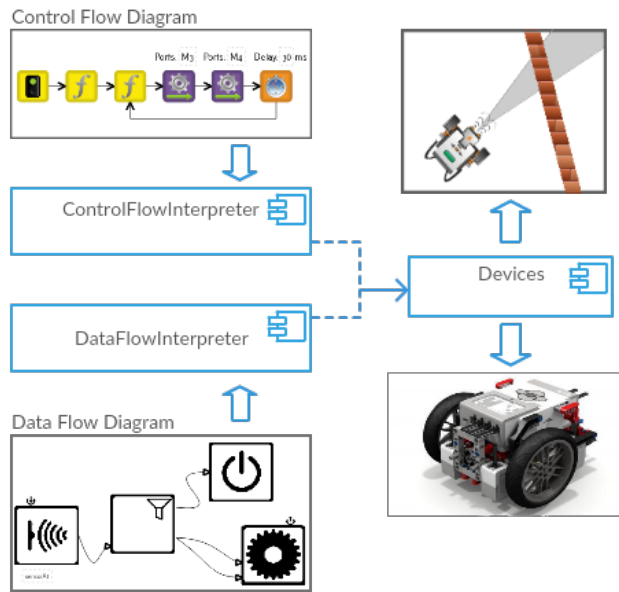


Figure 9: The general architecture of the system

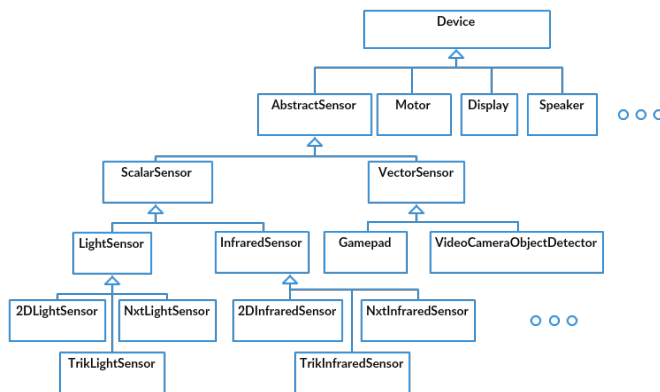


Figure 10: Partial architecture of devices used in dataflow interpreter

next, those are blocks without incoming flows. After all that done preparation phase is complete and diagram starts being interpreted.

Interpretation process is not as straightforward as in most asynchronous dataflow environments. Usually components of dataflow diagram are executed concurrently, on different threads, processes or even machines (that is actively exploited, for example, by Microsoft Robotics Developer Studio where dataflow diagram is deployed into a number of web-services). That is a pretty convenient way to invoke dataflow diagrams on a powerful hardware, but not a case when we talk about embedded devices. In our case we deal exactly with embedded devices (Lego NXT, EV3, TRIK, Arduino controllers), so we propose here another way of executing dataflow diagrams. The main idea is to introduce global message queue and event loop for messages

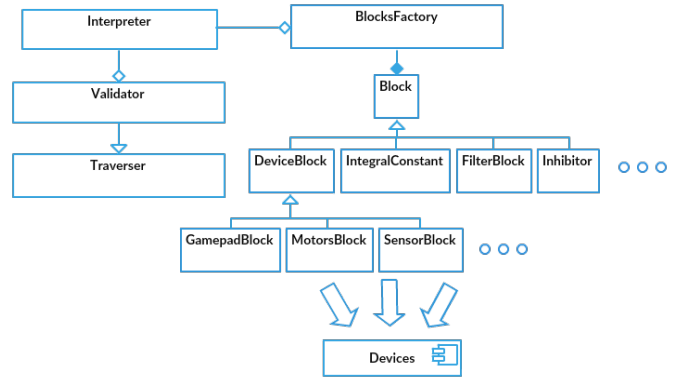


Figure 11: The general architecture of dataflow interpreter plugin

processing. When token is published by some block it is enqueued into messages queue and waits for its turn to be delivered to subscribers (fig. 12). In fact thus we *flatten* the execution, convert concurrent way of dataflow interpretation to a pseudo-concurrent one where we schedule invocation order on our own. It must be noted that this mechanism is similar to events propagation system of Qt framework. That is actively exploited in our implementation, where message processing is completely performed by *QEventLoop* class and tokens delivering is done by Qt signal/slot system in *QueuedConnection* mode.

Flat execution of dataflow diagram poses a number of small problems, one of them will be discussed here. Input device blocks (for example blocks publishing tokens from ultrasonic sensors) are constantly emitting tokens to subscribers. Subscribers transmit tokens to a next one (possibly in modified state) and so on. Thus there appears a chain of data processing. In our language that chain can activate control flow ports of blocks “reviving” them, so the control flow model is implicitly supported in our language (this is important in educational reasons). If later in this chain same input device block will be met then execution will come in a counterintuitive way. Such conflicts are ruled out with a simple heuristic that among all the blocks sharing one physical device only one can be active and that is the last activated one. Thus when the execution token comes into some device block it immediately “deactivates” conflicting ones. Other problems like messages balancing (in case when some block “flooding” the whole messages queue) will not be discussed here.

The last thing we should remark here is the presence of *Fork* block in our language that usually is not provided by dataflow languages. Flattened model seems to work well on embedded devices, but sometimes users still need to use concurrent execution (for example for executing layers in Subsumption architecture). For that reason *Fork* block is introduced, it forks the execution into a number of platform-specific execution units (for example *pthreads* on UNIX or *tasks* on NXT OSEK). This block can be regarded as low-level control of execution process. It should be also marked

that this block almost has no sense in interpretation mode (because execution itself is performed on desktop machine with only sending primitive commands to robot), but will be very useful in future works when autonomous mode will be introduced.

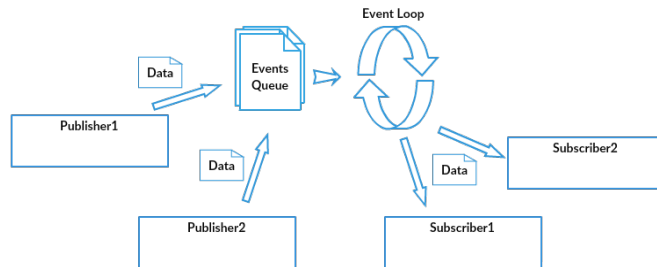


Figure 12: Proposed mechanism of pseudo-concurrent dataflow interpretation

## 7. Conclusion and Discussion

In this work we presented the prototype of dataflow language for programming different robotic kits (LEGO MINDSTORMS NXT, LEGO MINDSTORMS EV3, TRIK). The system provides ability to interpret diagrams on 2D- and 3D-simulators and real robotic devices. Here, we also propose an approach for executing dataflow diagrams on embedded devices. The language implicitly supports control flow model for educational purposes. It is also convenient for expressing typical robotic controllers architectures which is demonstrated on example.

The implemented system can be regarded as a platform for future investigations. First of all autonomous mode of work will be implemented. That will be done through code generation into a number of textual languages already supported by TRIK Studio (NXT OSEK C for Lego, bytecode for EV3, JavaScript, F# [32] and Kotlin for TRIK). We are also interested in academical research. First of all a formal semantics of our language should be expressed for applying various formal methods of program analysis. Another branch of research will be directed into a DSM-branch, here we want to consider an ability of dynamic language meta-model generation from specifications of available modules of robotics middleware (like ROS [33] or Player [34]).

## References

- [1] "IEEE International Conference on Robotics and Automation," 2016. [Online]. Available: <http://www.icra2016.org/>
- [2] "International Conference on Intelligent Robots and Systems," 2016. [Online]. Available: <http://www.iros2016.org/>
- [3] "IEEE Symposium on Visual Languages and Human-Centric Computing," 2016. [Online]. Available: <https://sites.google.com/site/vl-hcc2016/>
- [4] O. Banyasad, "A visual programming environment for autonomous robots," Master's thesis, DalTech, Dalhousie University, Halifax, Nova Scotia, 2000.
- [5] J. Simpson, C. L. Jacobsen, and M. C. Jadud, "Mobile robot control," *Communicating Process Architectures*, p. 225, 2006.
- [6] J. Simpson and C. L. Jacobsen, "Visual process-oriented programming for robotics," in *CPA*, 2008, pp. 365–380.
- [7] J. C. Posso, A. T. Sampson, J. Simpson, and J. Timmis, "Process-oriented subsumption architectures in swarm robotic systems," in *CPA*, 2011, pp. 303–316.
- [8] J. P. Diprose, B. A. MacDonald, and J. G. Hosking, "Ruru: A spatial and interactive visual programming language for novice robot programming," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 2011, pp. 25–32.
- [9] "NXT-G quick programming guide," 2013. [Online]. Available: <http://www.legoengineering.com/nxt-g-quick-guide/>
- [10] "All about TRIK: TRIK Studio," 2016. [Online]. Available: <http://blog.trikset.com/p/trik-studio.html>
- [11] "ROBOLAB quick guide," 2013. [Online]. Available: <http://www.legoengineering.com/robolab-quick-guide/>
- [12] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.
- [13] "LabVIEW System Design Software - National Instruments," 2016. [Online]. Available: <http://www.ni.com/labview/>
- [14] "Simulink - Simulation and Model-Based Design," 2016. [Online]. Available: <http://www.mathworks.com/products/simulink/>
- [15] "MINDSTORMS EV3 - Products," 2016. [Online]. Available: <http://www.lego.com/en-us/mindstorms/products/>
- [16] "ScratchDuino — Magnetic Robot Construction Kit," 2016. [Online]. Available: <http://www.scratchduino.com/>
- [17] A. Kuzenkova, A. Deripaska, K. Taran, A. Podkopaev, Y. Litvinov, and T. Bryksin, "Sredstva bustroi razrabotki predmetno-orientirovannykh resheniy v metaCASE-sredstve QReal," *St. Petersburg State Polytechnical University Journal*, p. 142, 2011 (in Russian).
- [18] A. Kuzenkova, A. Deripaska, T. Bryksin, Y. Litvinov, and V. Polyakov, "QReal DSM platform — An Environment for Creation of Specific Visual IDEs," in *ENASE 2013 — Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*. Setubal, Portugal: SciTePress, 2013, pp. 205–211.
- [19] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.
- [20] M. Proetzsch, T. Luksch, and K. Berns, "The behaviour-based control architecture ib2c for complex robotic systems," in *KI 2007: Advances in Artificial Intelligence*. Springer, 2007, pp. 494–497.
- [21] B. Erwin, M. Cyr, and C. Rogers, "Lego engineer and robolab: Teaching engineering with labview from kindergarten to graduate school," *International Journal of Engineering Education*, vol. 16, no. 3, pp. 181–192, 2000.

- [22] J. M. Gomez-de Gabriel, A. Mandow, J. Fernandez-Lozano, and A. J. Garcia-Cerezo, "Using lego nxt mobile robots with labview for undergraduate courses on mechatronics," *IEEE Trans. Educ.*, vol. 54, no. 1, pp. 41–47, 2011.
- [23] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2007.
- [24] S. H. Kim and J. W. Jeon, "Programming lego mindstorms nxt with visual programming," in *Control, Automation and Systems, 2007. IC-CAS'07. International Conference on.* IEEE, 2007, pp. 2468–2472.
- [25] J. Simpson and C. G. Ritson, "Toward process architectures for behavioural robotics," in *CPA*, 2009, pp. 375–386.
- [26] J. H. Connell, "A colony architecture for an artificial creature," DTIC Document, Tech. Rep., 1989.
- [27] R. C. Arkin, "Motor schema based navigation for a mobile robot: An approach to programming by behavior," in *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, vol. 4. IEEE, 1987, pp. 264–271.
- [28] J. K. Rosenblatt, "Damn: A distributed architecture for mobile navigation," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 339–360, 1997.
- [29] D. V. Koznov, *Osnovy vizual'nogo modelirovaniya*. [Fundamentals of Visual Modeling] Binom. Laboratorija znaniy, Internet-universitet informacionnyh tehnologij, 2008 (in Russian).
- [30] "The Programming Language Lua," 2016. [Online]. Available: <https://www.lua.org/>
- [31] E. Rohmer, S. P. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on.* IEEE, 2013, pp. 1321–1326.
- [32] A. Kirsanov, I. Kirilenko, and K. Melentyev, "Robotics reactive programming with F#/Mono," in *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia.* ACM, 2014, p. 16.
- [33] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.
- [34] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1, 2003, pp. 317–323.

# Programming languages segmentation via the data mining software “ShaMaN”

Tatiana Afanasieva  
Dept. of Information systems  
Ulyanovsk state technical University  
Ulyanovsk, Russia  
tv.afanasjeva@mail.com

Anton Efremov  
Dept. of Information systems  
Ulyanovsk state technical University  
Ulyanovsk, Russia  
t-rain@mail.ru

Sveta Makarova  
Dept. of Information systems  
Ulyanovsk state technical University  
Ulyanovsk, Russia  
makarovasvetlana2025@gmail.com

Denis Shalaev  
Dept. of Information systems  
Ulyanovsk state technical University  
Ulyanovsk, Russia  
melges73@gmail.com

**Abstract**— The article is devoted to the problem of programming language choosing. We conducted a research which purpose is to identify the most promising and comfortable programming language for work and study. The research object is software product, namely 9 of the most popular programming languages. This problem is urgent for all subjects of software engineering in solving problems of planning, optimization and analysis of the market.

There is currently a high popularity of IT-technologies that means high-speed trends changing. To be a successful participant of the IT-community it is necessary to constantly analyze this industry. We offer software for data mining named "ShaMaN". Its peculiarity is analyzing data from any domain, including programming languages.

**Keywords**— *programming language, segmentation, clustering, fcm-clustering, research*

## I. INTRODUCTION

It is impossible to imagine the modern world without the Internet, total automation, networked gadgets and virtual reality. More and more spheres of life become dependent on IT-technologies. For example, nowadays you can pay for services directly through the mobile app without getting up from the bed, you will need just a couple of seconds for it. You have an opportunity to monitor your home and manage its energy supply remotely. The popular concept "Smart house" is also bound to the possibility of the development of Internet technologies.

The information technology demand is increasing every year at a rapid pace. According to the research "Review and assessment of the prospects of development of the world and Russian market of information technologies", conducted by the independent research firm IDC commissioned by the Moscow Stock Exchange and Russian Venture Company, it is the most dynamic segment of the software. This explains the popularity of the profession programmer in the labor market.

The figures speak for themselves. Programmer occupies 6th place in Russia with the size of salaries. The demand is always higher than the offer in IT-industry. The average salary is 70,000 rubles. But there are special requirements to programmers: they must always keep track of tendencies. IT-industry is dynamic, rapidly changing area, which requires from its members the flexibility and ability to quickly learn other programming languages. Therefore, both beginners and experienced professionals are constantly faced with the question: "Which programming language to choose?"

## II. RESEARCH SUBJECT

The problem of finding an answer to the question "What programming language to learn?" is the goal of this research "Segmentation of programming languages". We are interested in finding the most comfortable and perspective at the same time language to start our professional activities. Language is comfortable if the efforts of its studying and job searching are comparable to salary.

Thus the purpose of this research is to identify the most comfortable and, at the same time, perspective programming language for running. To achieve this goal it is necessary to segment programming languages according to the stated quality. This research could be divided into 6 tasks:

- Research object analysis and the identification of its characteristics.
- Selecting attributes of segmentation according to the purpose to the research.
- Building a set-theoretic model.
- Data sources searching.
- Determination of a data model and filling the database.
- Data segmentation via the program "ShaMaN".

- Searching answers to the question via segmentation results.

Tasks determining allows to clearly delineate responsibilities among the participants of the research as it increases the probability of getting the expected result.

### III. PROGRAMMING LANGUAGES AS THE RESEARCH OBJECT

49 programming languages are allocated on web-service for IT-projects hosting «GITHUB». Each of them can be characterized by the popularity among programmers, field of use, the average wage, age, popularity in the labor market, level, category, etc. But only 3 attributes are interesting from the standpoint of research: the average wage, the popularity on the labor market and the popularity among programmers. The set of high levels of these three attributes is the definition of programming language.

Programmer wage is a determining factor in the profitability of its study. The data source for its evaluation is the results of the survey of 26,086 programmers from 157 countries. The survey was conducted by «Stack Overflow» among users of their web-site in order to assess trends in the IT-community. On the company's website presents data for the United States of America, Eastern and Western Europe are presented. The programmer average wage will be measured in dollars per year and denoted as  $x_i$ , where  $i \in [1..n]$ ,  $n \in N$ ,  $n$  – the programming language quantity.

Another attribute for programming languages analyzing is their a popularity among programmers. At first glance it is not clear for the review, but it is important. The higher its value, the more opportunities for programmers to find supporting information in the Internet. Formally, popularity is the percentage of respondents of the «Stack Overflow» survey engaged in development using a particular programming language. It is signed for  $y_i$ , a unit of measurement -%.

Selecting popularity on the labor market as an attribute for the segmentation of programming languages is evident. The demand on the labor market is a direct reflection of the economic situation and proves the prospects of the profession. Collection of data to research the programming languages on this attribute was made via web-services occupation searching: HeadHunter.ru for Eastern Europe and recruit.net for the US and Western Europe. Formally, popularity on the labor market is the number of vacancies we found searching a job for programmer of different languages. It is measured in units and is indicated as  $z_i$ .

The set-theoretic model of research will be:

$$L = \{X, Y, Z\} \quad (1)$$

where  $L = (l_1, l_2, \dots, l_n)$ ,  $L$  – programming languages;

$L \in \{\text{Java Script, SQL, JAVA, C\#, php, Pethon, C++, C, Node.js, AngularJS, Ruby, Objective-C}\}$ ;

$X = (x_1, x_2, \dots, x_n)$ ,  $X$  – the average wage [\$ / year];

$Y = (y_1, y_2, \dots, y_n)$ ,  $Y$  – popularity among programmers [%];

$Z = (z_1, z_2, \dots, z_n)$ ,  $Z$  – popularity on the labor market [units].

### IV. TASKS FORMALIZATION

To answer the main question of the research it was decided to segment programming languages on the main attributes, on the average income in the context of individual regions (geographical segmentation) and in the context of time periods (temporal segmentation). Below are the formal records of the planned phases of the segmentation.

The geographical segmentation by IT-specialist average income:

$$S = \text{segmentation}(L\{X\}, \text{country}_i) \quad (2)$$

where  $S = (S_1, S_2, \dots, S_m)$ ,  $S$  – segments of programming languages, which was received as a result of its segmentation for country<sub>i</sub> by average wage  $X$ ;

country<sub>i</sub> – is country for segmentation, country<sub>i</sub>  $\in \{\text{CHIA, West Europe, East Europe}\}$ ;

$i \in [1, k]$ ,  $k \in N$ ,  $k$  – the number of countries;

$m \in N$ ,  $m$  – the number of segments.

Tine segmentation on languages popularity among programmer has a model:

$$S = \text{segmentation}(L\{Y\}, \text{time}_i) \quad (3)$$

where  $S = (S_1, S_2, \dots, S_m)$ ,  $S$  – segments of programming languages, which was received in results of its segmentation in the moment of time<sub>i</sub>;

time<sub>i</sub> – time segmentation interval, time<sub>i</sub>  $\in \{2013 \text{ г., } 2014 \text{ г., } 2015 \text{ г.}\}$ .

Segmentation on main attributes:

$$S = \text{segmentation}(L\{X, Y, Z\}) \quad (4)$$

где  $S = (S_1, S_2, \dots, S_m)$ ,  $S$  – segments of programming languages, which was received in results of its segmentation on main attributes.

#### 5. PROGRAM “SHAMAN” AS THE MAIN RESEARCH TOOL

Program “ShaMaN” was developed in 2014 for making researches. The main appointment is processing of statistical information automation, as well as an intellectual analysis of this information. There is the certificate of state registration of the computer №2014661216 for the program “ShaMaN”, which was used in this research. Also this program was declared as the winner of the contest of scientific and technical creativity of the youth of the Volga Federal District in 2015

##### A. Program features

Program “ShaMaN” allows to group the data sampling objects using a number of clustering methods (fcm-clustering and clustering based on the spanning tree). Production data can be automatically loaded into the system from SQLite databases, MS Access, MS SQL Server, as well as MS Excel spreadsheets, MS Word and web-sites. The main feature of the system is type independence of the type of input data and structure. When you boot into the system, they are normalized by using methods of calculation distances between objects. The results of clustering data displayed on the screen, a graph

is constructed for clarity. These results can be saved as Excel 1997-2003 (.xls) file.

### B. Architecture

The system consists of four parts: a data loading, data processing, reporting and user interface. Special interfaces are used in this program to ensure the interaction of each part. Each new module is a separate part which should inherit the appropriate interface. Thus, the system can be easily expanded with new functionalities. There is the class diagram on the figure 1.

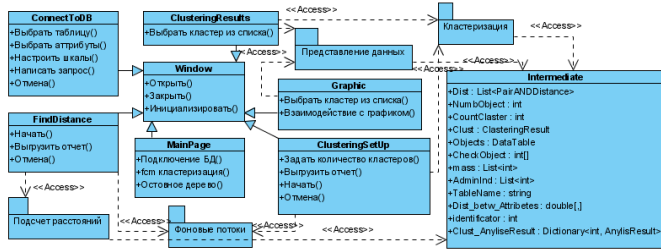


Fig. 1. "ShaMaN" class diagram

### C. Mathematical provision

Mathematical provision includes algorithms for calculating distances between objects and attributes, fcm-clustering and clustering based on the spanning tree.

#### 1) Accounting distances between objects

Measurements of the distance between objects show how far objects are located from each other in space. They have the following properties:

- the properties of distance measurements for different types of scales;
- continuity  $\delta(x,y)$  – continuity function;
- symmetrical  $\delta(x,y)=\delta(y,x)$ ;
- normalization  $0 \leq \delta(x,y) \leq 1$ , then  $x=y$ , to  $\delta(x,y)=0$ ;
- invariance  $\delta(x,y) = \delta(\varphi(x),\varphi(y))$ ;
- properties triangle  $\delta(x,z) \leq \delta(x,y)+\delta(y,z)$ .

To determine the distance between the objects you need to find the distance between them in all their attributes, and then the resulting Euclidean distance:

$$\delta = \sqrt{\delta_c^2 + \delta_n^2 + \delta_n^2} \quad (5)$$

#### 2) Accounting distances between attributes

The method of counting the distance between features is defined according to what they measured in any scales. For example, if both signs are presented in strong, the distance is determined by the formula [1]:

$$\delta_{cc} = 1 - r \quad (6)$$

where  $r$  – linear correlation coefficient module:

$$r = \frac{M[(x-Mx)(y-My)]}{[\delta_x \cdot \delta_y]} \quad (7)$$

where  $M$  – expected value;

$\delta_x, \delta_y$  – standard deviation.

If the scale is weak (order or names), the distance is defined as the Kendall-Kemeny measure:

$$\delta_{nn} = \delta_{nn} = \left(\frac{1}{C_m^n}\right) \sum_{x,y} \delta_*(x,y) \quad (8)$$

where  $\delta_*(x,y)$  – the distance between the objects measured in the order scale or in the names scale;

$m$  – number of objects;

$n$  – number of comparable attributes – 2;

$C_m^n$  – determine by the formula:

$$C_m^n = \frac{m!}{n!(m-n!)} \quad (9)$$

Consider the second case, when the attributes are of different types: a strong scale compared with weak [2]. In this case it is necessary to resort to the methods of strengthening the weak scale and weakening the strong. The final distance is determined by the formula:

$$\delta_{\uparrow\downarrow} = \frac{[\delta_{\uparrow\uparrow} + \delta_{\downarrow\downarrow}]}{2} \quad (10)$$

where  $\delta_{\uparrow\uparrow}$  – strengthening «digitizing»;

$\delta_{\downarrow\downarrow}$  – weakening «association».

#### 3) FCM-clustering

FCM-clustering algorithm purpose is automatic clustering set of objects that are specified feature vectors in feature space. In other words, this algorithm determines the clusters and allocates the objects between them. Clusters are represented by fuzzy sets, moreover, the boundaries between the clusters are also unclear. [2]

This algorithm is based on the determination of the centers of clusters, followed by calculation of degree of membership of each object to the cluster. These steps are repeated until the difference between the matrices of degrees and supplies the current phase will lasts not less than a certain parameter  $\varepsilon$ .

Membership degree is calculated by the formula [3]:

$$u_{ij} = \frac{1}{\sum_{k=1}^C \left( \frac{\|x_i - c_j\|}{\|x_i - c_k\|} \right)^{\frac{2}{m-1}}} \quad (11)$$

Cluster centers are defined by the formula:

$$c_j = \frac{\sum_{i=1}^N u_{ij}^m x_i}{\sum_{i=1}^N u_{ij}^m} \quad (12)$$

where  $N$  – number of objects;

$C$  – number of clusters;

$m$  – any number more than 1 (usually 1,5);

$c_j$  – value of  $j$ -attribute of cluster center;

$x_i$  – value of  $i$ -ro attribute object  $x$ .



#### 4) Clustering algorithm based on spanning tree of minimum length

This clustering algorithm is based on determining the minimum spanning tree length and subsequent removal of  $n-1$  edges with a maximum length, where  $n$  is the number of clusters (figure 2).

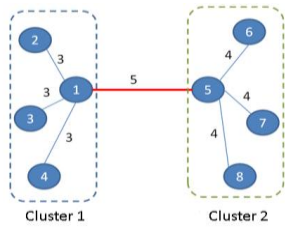


Fig. 2. Clustering algorithm illustration

Figure 2 shows the minimum spanning tree. By removing the link between nodes 1 and 5 with a length of 5 (maximum edge length), we obtain two clusters  $\{1, 2, 3, 4\}$  and  $\{5, 6, 7, 8\}$ .

Thus, the realization of searching the spanning tree is the main part of algorithm. The "greedy" algorithm Dijkstra Prima has been chosen for it [4]. "Greedy algorithms" operate using at each moment only a part of input data and taking the best solution based on this part. At each step you need to consider the set of edges that allow connection to the already constructed part of the spanning tree and choose from them the edge with the lowest weight. We could obtain a spanning tree repeating this procedure. The algorithm is presented in block diagram form in Figure 3.

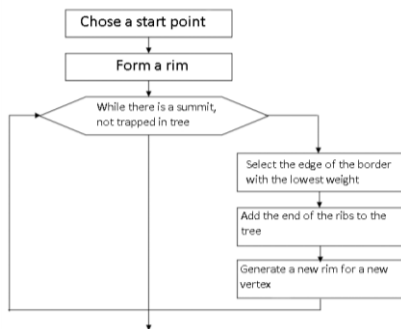


Fig. 3. Block diagram of the algorithm Dijkstra-Prim

Described clustering algorithm has been modified since it had a drawback: when a situation of "fan", which is shown in figure 4, in one of the clusters got only one object. In this case, the object number is 2.

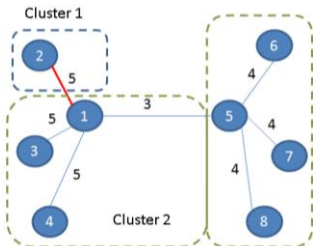


Fig. 4. The "Fan" in the graph

This problem was solved by adding additional processing. When a "fan", as shown in Figure 3, was found the algorithm redirected items 2, 3, 4 with the same distance to any object from the "fan". Thus the assembly of clusters (DeterminateCluster method) Clusters 1 and 2 are well defined (fig. 5).

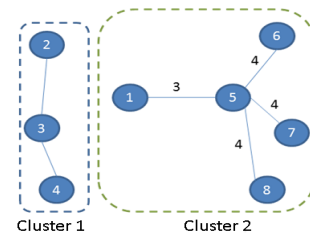


Fig. 5. Solving the "fan" problem

#### D. Information provision

"ShaMaN" system allows to process information regardless of the structure and data types. The system loads data from database files with one of the following extensions: mdf, mdb, db, db3, accdb. Just as input table located on web-pages, Word or Excel files may act. The output of the system are the Excel 1997-2003 files with the extension xls.

#### V. TIME SEGMENTATION

The temporal segmentation was held to analyze the dynamics of changing programming language popularity among programmers. It's allowed to identify trends in using git repositories by programmers. As the source of data was used the web-site github.info. GitHub - is a relatively new resource that examines the 2.2 million active repositories on GitHub. The resource analyzes public repository, which may cause deviation of the results in the direction of open source technologies.

Graphic in figure 6 is the result of time segmentation of programming languages. It was based on trends of using repository on GitHub. The graphic shows the dynamics of changes in the number of active repositories for 2012, 2013 and 2014.

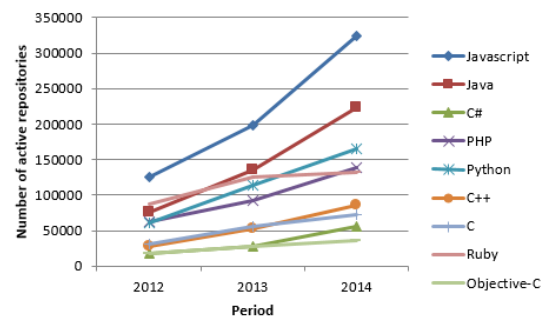


Fig. 6. The time segmentation result

Interpretation of the time segmentation results allowed to draw the following conclusions:

- The number of active repositories is increasing for all programming languages. This is due to the overall

dynamics of growth IT-industry growth in general and software in particular segments.

- The most popular programming languages are Java and Javascript. They have the highest pace of growth.

#### VI. GEOGRAPHIC FEATURES SEGMENTATION

Programming languages were considered in three geographic levels: the United States, Western Europe on the example of the UK, Eastern Europe on the example of Russia and CIS countries. In this aspect it was decided to explore the demand among employers. It is evident that the demand on the labor market is a direct reflection of the economic situation. It can talk about the prospects of the profession. Data for programming languages analyzing on this attribute was taken from web services for searching work: HeadHunter.ru Eastern Europe, namely Russia and the CIS countries, and recruit.net for the US and Western Europe by the example of Great Britain. Formally, the popularity among employers is the number of vacancies for programmers in this language. In order to analyze the relevance of different programming languages among employers in different countries have been extracted data for each programming language and are segmented on a geographic basis.

The chart was constructed as a result of the segmentation. The diagram (figure 7) shows the relevance of demand in programming languages in the US, UK, Russia and CIS countries.

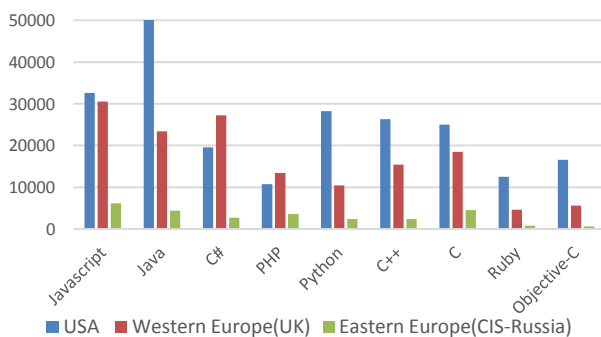


Fig. 7. Geographical segmentation

Interpretation of the results allowed to make the following conclusions:

- The United States is the leader in the number of vacancies in 7 cases out of 9. Only in the case of C# and php Western Europe is ahead. This is due to the fact that the US is the world leader in the IT-sector.
- The most demanded language in the United States - Java; in Western Europe and the CIS - Javascript.

Data were also analyzed on the average labor payment on programming languages and performed segmentation by geography. The results were presented separately for each country in the form of graphs, presented in the figures.

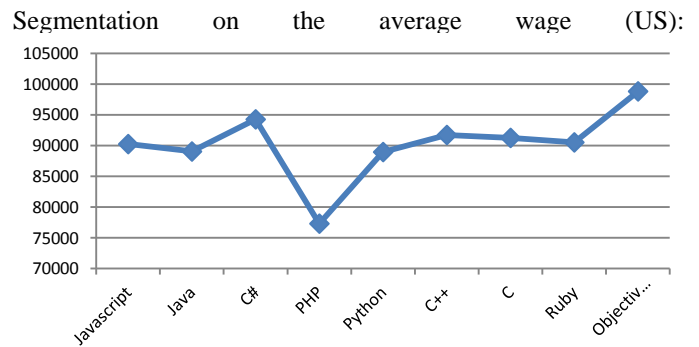


Fig. 8. Average labor payment (US)

Segmentation on the average wage (Western Europe):

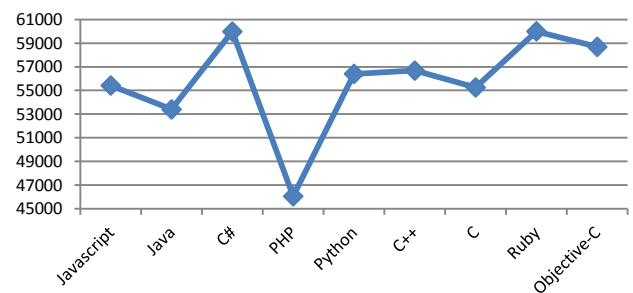


Fig. 9. Average labor payment (Western Europe)

Segmentation on the average wage (Eastern Europe):

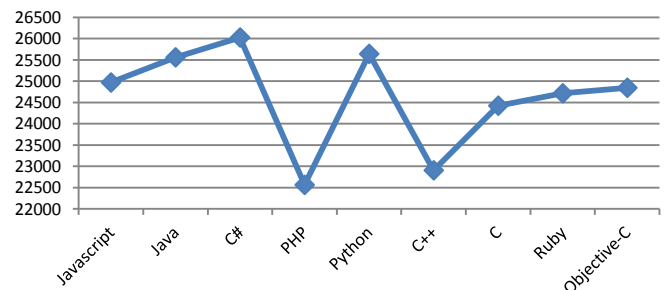


Fig. 10. Average labor payment (Eastern Europe)

Interpretation of the results allowed the following conclusions:

- The average wage in the US in all programming languages is higher than in Western and Eastern Europe. US occupy leading positions in the IT-sector and therefore have higher average salaries in this sector.
- Most paid programming language in US is Objective-C.
- Most paid programming languages in Western Europe are Ruby and C#.
- Most paid programming languages in Eastern Europe are C# and Python.



- The level of wages in Eastern Europe is the lowest of the ones under consideration. This region does not occupy high positions in the sector of IT-industry.
- The least paid programming language in all the above areas is php.

## VII. THREE ATTRIBUTE SEGMENTATION

The final stage of the research is the segmentation of programming languages on three grounds, which will highlight the classes according to the degree of comfort and the prospects for study and work. Segmentation was carried out using the above program "ShaMaN". Clustering method based on spanning tree of minimum length was used.

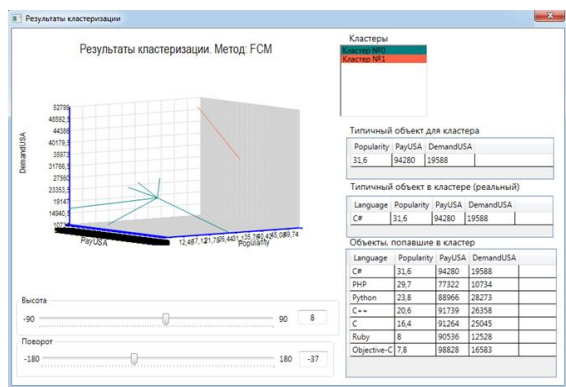


Fig. 11. The “ShaMaN” clustering window

As a result of the programming languages clustering in the United States on three attribute (popularity among programmers, the average wage in the US and demand among employers) has been allocated 2 cluster. In the second cluster were languages with the highest performance values: Java and Javascript. For clarity, the results are summarized in Table 1 and Table 2, the characteristics values which are higher than the average, are highlighted. This leads to the conclusion that Java and Javascript are the most promising programming languages in the United States. The most comfortable programming languages are C#, C++ and C (2 of 3 characteristics have a value higher than the average).

TABLE I. FIRST CLUSTER

Programming language	Popularity	Payment	Demand
C#	31,6	94 280	19 588
PHP	29,7	77 322	10 734
Python	23,8	88 966	28 273
C++	20,6	91 739	26 358
C	16,4	91 264	25 045
Ruby	8	90 536	12 528
Objective-C	7,8	98 828	16 583

TABLE II. SECOND CLUSTER

Programming language	Popularity	Payment	Demand
Javascript	54,4	9 0 259	32 579
Java	237,4	89 054	52 799

Eastern and Western Europe were analyzed. In both cases the second cluster consists of objects – extrema, which attributes values are maximal or minimal. The results allowed to make the following conclusions:

- Javascript is the most promising programming language in Eastern Europe, comfortable - Java, C#, php are the most comfortable.
- C#, Java, Javascript are the most promising programming languages in Western Europe, php, Python, C ++, C are the most comfortable

The results of clustering programming languages within the geographical entities are summarized in Table 3. For each research object the number of points that corresponds to the number of times was estimated when the value of the characteristics of a particular programming language was above an average. Taking into account the final amount of points it is possible to allocate a group of programming languages. It will allow to answer the main research question: which language is the most comfortable and the most promising for studying and working?

TABLE III. CLUSTERING RESULTS

Cluster	Programming language	Popularity	Payment	Demand	Sum
№1	Java	3	3	3	9
№2	Javascript	2	3	3	8
	C#	3	2	3	8
№3	PHP	0	1	3	4
№4	Python	2	1	0	3
	C++	2	1	0	3
№5	C	1	3	0	4
№6	Ruby	3	0	0	3
	Objective-C	3	0	0	3

6 groups of programming languages were determined:

- Java is a programming language - the star. It is popular, highly paid and in demand worldwide. However, to stay in the trend, to deal with competition the programmer has to constantly work on improving his skills.
- Javascript and C # are forward-looking and comfortable programming languages that are worth exploring, regardless of the country.

- Python and C ++ are comfortable programming languages in most countries: there is a low competition at high pay.
- C is a comfortable programming language in demand around the world and it is well paid.
- Ruby and Objective - C are not in demand, are not popular, but they are well-paid. These languages go out of your comfort zone, because it requires effort to find a job. But if a programmer is successful he is rewarded with higher wages.
- php is unpromising programming language, which involved all, but no one needs it. Most likely due to the popularity it he enjoyed before, now on the market there is more staff number of frames with php knowledge. And some of them retrained to a different language programmers, and some remained with php.

#### VIII. CONCLUSION

This article considers the process of finding the answer to the question "What programming language should I learn?". A research was conducted to identify the most comfortable and, at the same time, promising programming language. This research has provided an experimental data for testing of the program "ShaMaN".

After analyzing the object of research and identifying its characteristics the following attributes for segmentation were

selected: popularity and demand of the programming language, as well as the average wages of a specialist. After building the set-theoretic model and obtaining data necessary for segmentation and clustering studies have been conducted. The research allowed to allocate 6 groups of programming languages, as well as to identify the programming language that best suits the objective of the study.

Java has been recognized as the most preferred programming language according to the objective of the study. This programming language is popular, highly paid and in demand worldwide. However, Javascript, and C # is not far behind the leader. This forward-looking and comfortable programming languages is worth exploring, regardless of the country.

#### REFERENCES

- [1] A. Yurkin Zadachnik po programmirovaniyu [Book of problems in programming]. – SPb.: Piter, 2002. 192 p. (rus).
- [2] N.G. Zagoruiko Prikladnyye metody analiza dannykh i znaniy [Applied methods of data analysis and knowledge]. – Novosibirsk: Izdatel'stvo instituta matematiki, 1999. 270 p. (rus).
- [3] N.G. Yarushkina Osnovy teorii nechetkikh I gibridnykh sistem [The fundamentals of fuzzy and hybrid systems theory]: tutorial / N.G. Yarushkina. – Moscow: Finansyistatistika, 2004. 320 p. (rus).
- [4] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica* 18, 263–270. 1997.

# Context-Based Model for Concern Markup of a Source Code

Mikhail Malevanny  
Rostov State University  
of Civil Engineering  
Rostov-on-Don, Russia  
Email: mmxforever@mail.ru

Stanislav Mikhalkovich  
Institute for Mathematics, Mechanics,  
and Computer Science in the name of I.I. Vorovich  
Southern Federal University  
Rostov-on-Don, Russia  
Email: miks@sfedu.ru

**Abstract**—In this paper we describe our approach to representing concerns in an interface of an IDE to make navigation across crosscutting concerns faster and easier. Concerns are represented as a tree of an arbitrary structure, each node of the tree can be bound to a fragment of code. We describe a model which specifies data structures and algorithms. Main goal is to keep concern tree consistent with evolving source code. The model is implemented in a tool, which supports different programming languages and integrates into different editors and integrated development environments.

## I. INTRODUCTION

During software development and maintenance developers usually work with several code fragments related to their current task or *concern*. Most concerns are crosscutting[1], which means that code related to it tends to be scattered across a number of files, or different places within one file. Repeated navigation between these code fragments requires a considerable time and effort[2]. These fragments form a "working set". Switching to another task requires investigating the source code and locating all fragments relevant to the task. Returning to the task after working on another one may take significant time.

A number of techniques address to this problem, such as Aspect-Oriented Programming[3], Feature-Oriented Programming[4] [5] [6], Delta-Oriented Programming[7], Subject-Oriented Programming[8]. Most of them are intended to explicitly separate concerns into a number of modules and provide different mechanisms of composition of these modules. It often requires significant changes in the source code to use one of these techniques.

Other methods provide support of concerns by adding new tools to an IDE, such as virtual files[9] [10] [11] colour markup[12] without changing the source code. These tools are often designed for only one IDE and depend on its infrastructure and thus are limited to only few languages, supported by the IDE. Another common limitation is low tolerance of changes in the source code. When the code is modified some code fragments may be lost.

Many of these tools are limited to only one programming language, while large software projects are often developed in several languages, including DSL-languages and markup languages, and code fragments related to a concern may be scattered across files in different languages.

We are currently developing an approach [13] intended to mitigate the problems of navigation across the code and switching between different tasks. The approach doesn't require any changes to the source code. It defines a notion of a concern as a tree-like structure, consisting of sub-concerns and code fragments. Similarly to ConcernMapper[14] it displays a concern tree in an IDE as a toolbox and allows one to quickly locate fragments in the source code. Unlike most other tools it and may be used in different IDEs and allows one to work with code in different languages. Another goal is robustness, which allows working with the code being actively developed keeping concern tree consistent with the code.

## II. MODEL

We present a model our approach is based on. It uses lightweight parsers to analyze source text and to create parse tree which will be used later. The model defines the data being stored in the concern tree. And finally, it defines algorithms to search the code fragments in a modified source code.

### A. Lightweight parsing

The model is common for different languages. To minimize dependency on IDE infrastructure we use lightweight parsing to analyze the source code and build parse tree, which contains information about significant entities in the code. Lightweight parsers can recover from errors and produce parse tree for code with errors or incomplete code, which is important while the code is being modified.

Adding support of another programming language requires development of a lightweight parser for this language. Lightweight parsers are simple and easy to develop using our DSL language LightParse. For most languages it takes only about 10-30 lines of text to express important language features and produce a lightweight parser. The parser is able to analyze source code and build a simple parse tree with only nodes, corresponding to these language features. Any other parts of source code (e.g. method bodies) are skipped. Saving information about an entity in the source code is available for all entities returned by the parser. The more detailed parse tree the parser produces – the more entities can be saved in the concern tree, however development of the parser may require more time.

Lightweight parsers produce a lightweight parse tree. Nodes of the tree have type, name and location in the source code. Node name consists of several tokens; one of them may be marked as important. For example method name consist not only of one identifier – name, which is marked as important, but also includes parameter names and types, access modifiers, return value type and so on.

An example of a lightweight parser is given in subsection II-C. Lightweight parsing is described in our paper [15] in more detail. The paper provides examples of lightweight parser grammar. More examples may be found in GitHub repository of the tool<sup>1</sup> (files with extension “.lp”).

## B. Data

The approach is not limited to any specific programming language and therefore the information in the concern tree should be sufficient to support different languages. Also, we assume that the source code may change and the concern tree should possibly store some redundant data to find the code fragment after the code has changed.

Each code fragment in the concern tree stores next 5 items:

- Type.
- Header context. It may include entity name and any number of additional tokens.
- Outer context. It includes names and types of all parent nodes from the immediate parent to the root of the parse tree.
- Horizontal context. It consists of two subsets of names and types of preceding and subsequent sibling nodes.
- Inner context. It includes a subset of subnodes of current code fragment.

These items form *Context* of the node. Except for type, any other item may be empty.

**Type** is used to filter non-relevant nodes when searching for the code fragments. If a concern tree item is bound to a method, only methods should be considered, other nodes, e.g. classes, fields may be ignored.

**Header context** represents entity name and several additional tokens. In the following C# code example

```
public void visit(TreeNode t)
public void visit(Expression e)
```

both methods are named `visit`, but have different parameter types and names. Header context makes possible distinguishing overloaded methods and other entities with same names. Header context is represented as a list of tokens, where one token may be marked as important and it is considered as the **name** of the entity. Header context as well as name may be empty.

**Outer context** stores enclosing entities for the code fragment, such as classes and namespaces. In many languages there may be variables and methods with exactly same names, but defined in different classes or namespaces. An example is the implementation of one interface by different classes. In this

case it's necessary to save not only the name of the entity, but also the name of enclosing entities. In the following example

```
namespace N
{
    class C1 : IVisitor
    {
        public void visit(IVisitor v) { }
    }
    class C2 : IVisitor
    {
        public void visit(IVisitor v) { }
    }
}
```

both methods have same names and header contexts, but are defined in different classes. For example, outer context for the first method will include name and type of class `C1` and namespace `N`. Outer context for an entity is a list of Header contexts and Types for each enclosing entity starting from the immediate parent to the topmost entity in the source file.

Header context and outer context are sufficient for most programming languages, where all names are unique, at least in a certain scope. However, there is another class of languages, such as Yacc (grammar definition language), or markup languages, such as XML. In these languages there may be two entities with same name in same scope. Without additional information binding concern tree nodes to such entities is ambiguous. To handle these cases two different kinds of context were added to the model.

**Horizontal context** keeps nearest neighbors before and after the node. It consists of two sets of pairs (Header context + Type), one for preceding entities and one for subsequent entities. Following example is an excerpt from ANSI C grammar[16]:

```
selection_statement
: IF '(' expression ')'
  statement ELSE statement
...
;
```

There are two occurrences of `statement` in a subrule of a rule `selection_statement`. Their horizontal contexts are different: token `ELSE` and another non-terminal `statement` are located *after* the first occurrence of `statement` and *before* the second one. This information makes it possible to distinguish similar entities by their location among their neighbor entities.

It could have been achieved by saving an *index* of the entity. For example, first `statement` gets index 1 and second one gets index 2, but saving indexes is less tolerant to changes in the source text. Adding or removing entities in the beginning of a subrule invalidates indexes of all subsequent entities, but has almost no effect on horizontal context.

**Inner context** is intended to store subnodes of an entity. In some cases an entity can have empty name and may be distinguished from another one only by its content. For

<sup>1</sup><https://github.com/MikhailoMMX/AspectMarkup/tree/master/Parsers>

example, variable declaration sections in such language as PascalABC.NET [17] are unnamed, but they have different variables:

```
var
    X, Y : Double;
var
    Name, Address : string;
    Age : integer;
```

In this example there are two sections. It may be necessary to bind a concern tree node to a whole section. Horizontal context cannot be reliable in this case because it keeps only type and name, which is empty – changing their order will lead to incorrect result of the search. Inner context is a set of Header contexts and Types for some subnodes. In the example above saving only one subnode (i.e. variable name) is enough to distinguish these sections. Amount of subnodes to be saved as the inner context may vary.

Inner context for leaves of a parse tree may contain lines of source code. This may apply if the entity spans multiple lines in the source code (e.g. methods).

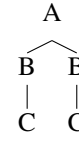
Inner and horizontal contexts may be empty if the entity has no neighbor nodes or subnodes. Otherwise it may be not necessary to store all neighbors or subnodes. Usually, a small amount of unique nodes is enough to distinguish similar entities. In many languages horizontal and inner contexts are a redundant information. However, using horizontal and inner contexts increases reliability of the search even with a code on a programming languages that normally don't need these two kinds of context. When the code has changed this information may be useful.

Let  $T$  is a parse tree node.  $Context(T) = (Name_T, Type_T, N_T, O_T, H_T, I_T)$  is a tuple of node Name, Type and its Header, Outer, Horizontal and Inner contexts described above. When a binding to the node  $T$  is added to the concern tree,  $Context(T)$  is saved.

*Name* and *Type* are strings. Header context  $N_T = (S_1, S_2, \dots, S_n)$  is a list of strings. Outer context  $O_T = ((N_1, T_1), (N_2, T_2), \dots, (N_n, T_n))$  is a list of pairs, where  $N_i$  is a Header Context and  $T_i$  is a type of an enclosing entity. Inner Context  $I_T = \{(N_i, T_i)\}$  is a set of pairs: header contexts and type of an entity. And Horizontal context  $H_T = (H_L, H_R) = (\{(N_i, T_i)\}, \{(N_j, T_j)\})$  is a pair of sets of header contexts and types of entities.

### C. Additional markup

Our approach is focused on finding code fragments without using any modifications of source code. Additional markup, such as comments with special keywords clutters the code if used frequently. However, in some cases it might be feasible to mark some places in the code with comments. First scenario is binding to code fragments in a file, which contains a lot of very similar entities. Some XML files may have such structure. In this example:



There are two nodes  $C$ , with equal contexts. Despite being subnodes of different parent nodes, their outer contexts are equal, because both parent nodes have same name. To handle this case it might require to save horizontal context for each parent node, which is not implemented in the model.

Another scenario is binding to code fragments in frequently modified code, where entities may undergo significant changes.

This kind of markup requires a lightweight parser which builds parse tree based on comments. Comments may define points and spans in the source code.

```
// ConcernBegin Serialization
...
// Concern SomePoint
...
// ConcernEnd Serialization
```

The code above shows an example of a markup with comments. *Concern Serialization* is a span and *SomePoint* is a single line marked with a comment.

Lightweight parser for this markup is simple and may work with source code in many languages. The only modification it may require to adapt the parser to a different language is changing comment start symbols. Here is a grammar of the lightweight parser written in LightParse:

```
%Extension "*"
Token Tk [[:IsLetterOrDigit:]]*|
           [[:IsPunctuation:]][:IsSymbol:]]
Token NewLine \r|\n|\r\n
Rule Program : [#Comment|Other]*
Rule Comment : "//" @CTk? @Tk+
Rule CTk:      @"ConcernBegin"
               | @"ConcernEnd"
               | @"Concern"
Rule Other :   Tk
             | NewLine
             | #error
```

## III. ALGORITHMS

There are two aspects of working with the concern tree: adding a node to the tree and searching the code fragment, related to the node. Both actions require a parse tree, which is provided by a lightweight parser. In the following part of the section we take into consideration only a subset of parse tree nodes whose type is equal to the type of an entity being saved or the one being searched. Given the  $T$  is a parse tree node to be saved in the concern tree, we consider a set  $Tree = \{T_i | Type_{T_i} = Type_T\}$ .

Next step is calculating a distance between  $T$  and every item  $T_i \in Tree$ .

### A. Calculating distances

Distance two tree nodes is a vector of distances between each component of a context for a given pair of nodes.

$Distance(T, T_i) = \overline{D}_i = (DName, DType, DN, DO, DH, DI)$ , where:

$$DType = \begin{cases} 1, & \text{if } Type_T \neq Type_{T_i}; \\ 0, & \text{if } Type_T = Type_{T_i}; \end{cases}$$

Distance for other part of context is calculated with functions  $LDistance$  and  $SDistance$ , described further below:

- $DName = LDistance(Name_T, Name_{T_i})$
- $DN = LDistance(N_T, N_{T_i})$
- $DO = LDistance(O_T, O_{T_i})$
- $DH = SDistance(H_T, H_{T_i})$
- $DI = SDistance(I_T, I_{T_i})$

Zero in each component of a vector  $\overline{D}$  means equality of corresponding parts of contexts of  $T$  and  $T_i$ . The higher these values – the less similar two parts of contexts are.

Calculating the distance for Name, Header context and outer context is based on a Levenshtein metric [18]. Levenshtein distance for two strings reflects the number of edits (insertions, deletions and substitutions) required to change one string into the other. Entity *Names* are just strings, however Header contexts are lists of strings. Levenshtein distance in this case is calculated similarly, but each edit is a deletion, insertion or substitution of a token. Weight of a substitution in this case depends on similarity of tokens and ranges between 0 (tokens are equal) to 2 (weight of insertion + weight of deletion) if two tokens have maximum possible edit distance between them. Distance between two outer contexts is calculated similarly. Each item of an outer context is a pair (Type, Header Context) and the weight of substitution depends on distance between to header contexts.

Calculation of edit distance is performed by overloaded functions  $LDistance$ .

Horizontal and inner contexts contain a subset of nodes and the distance is calculated as a number of subnodes present in  $T$  and absent in  $T_i$ .

Calculation of distance between sets is performed by function  $SDistance$

$$SDistance(I, I_i) = |I \setminus I_i|.$$

$$SDistance(H, H_i) = |H_L \setminus H_{iL}| + |H_R \setminus H_{iR}|.$$

### B. Saving information

Name, Type, Header and Outer contexts are required parts of a context and are saved always. Inner and Horizontal contexts are optional in some cases. To determine should they be saved or not and how much nodes they should contain we are looking for other nodes in the parse tree with similar Header Contexts.

Given the  $T$  is the parse tree node to be saved, we define two sets of parse tree nodes:

$$TreeL = \{T_i \mid O_{T_i} = O_T\}$$

$$TreeG = \{T_i \mid O_{T_i} \neq O_T\}$$

In other words, one subset consists of all neighbour nodes for  $T$  (Local scope) and other one - of all other nodes (Global scope).

After that we calculate two values:  $NearL$  and  $NearG$ .

$$NearL = LDistance(N_T, N_{T_i}) : T_i \in TreeL; \forall T_j \in TreeL, LDistance(N_T, N_{T_j}) \geq LDistance(N_T, N_{T_i}).$$

In other words, we find a distance between header contexts of  $T$  and the most similar node *within* the scope of a node  $T$ .

$$NearG = LDistance(N_T, N_{T_i}) : T_i \in TreeG; \forall T_j \in TreeG, LDistance(N_T, N_{T_j}) \geq LDistance(N_T, N_{T_i}) - \text{similar to } NearL, \text{ but outside of the scope of } T.$$

When  $NearG > 0, NearL > 0$  there are no other nodes with same header. In this case Inner and Horizontal contexts are optional and may be omitted. If  $NearG = 0, NearL > 0$  there are similar nodes with different outer context. Again, saving Inner and Horizontal contexts is optional, but may improve search results if the source file is modified. In case of  $NearL = 0$  saving inner and horizontal context is required.

The values  $NearL$  and  $NearG$  are saved within the concern tree and will be used for the search.

### C. Searching

A node in the concern tree keeps Context of some node  $T$ .

$$Context(T) = (Name_T, Type_T, N_T, O_T, H_T, I_T).$$

After some modifications were applied to the source file, target node may change as well. In some cases target node may be absent in the parse tree, if the code fragment related to the concern was removed. We do not address this case in our research and the tool is designed to always try to find target node or suggest a list of most similar entities.

The search begins with parsing a file and calculating edit distance  $\overline{D}_i = Distance(T, T_i) \forall T_i \in Tree$ .

Next step - checking if there is only one node in the tree, which is similar to the target node and therefore considered as the result of the search. It depends on values  $NearG$  and  $NearL$ .

If  $NearL > 0$ , then there was only one entity in the source file with Header context  $H_T$ . In this case if there is only one node  $T_i$  with similar Header context in the tree - it is returned as the result:

$$Result = T_i \in Tree :$$

$$LDistance(N_T, N_{T_i}) < \frac{Min(NearG, NearL)}{2};$$

$$\forall T_j \neq T_i \quad LDistance(N_T, N_{T_j}) > \frac{Min(NearG, NearL)}{2}$$

If  $NearL = 0$ , then there were other entities in the source tree, but only in the same scope as  $T$ . In addition to the condition above we can return  $T_i$  if it has minimal distance for Header, Inner and Horizontal contexts among all other nodes:

$$Result = T_i \in Tree : \forall T_j \neq T_i :$$

$$LDistance(N_T, N_{T_i}) \leq LDistance(N_T, N_{T_j})$$

$$LDistance(I_T, I_{T_i}) << LDistance(I_T, I_{T_j})$$

$$LDistance(H_T, H_{T_i}) << LDistance(H_T, H_{T_j})$$

These conditions are correct if  $NearG > 0$ . Otherwise there were other entities in the source file with same Header Context outside of the scope of  $T$ . In this case we add requirements



$LDistance(O_T, O_{T_i}) = 0$  and  $LDistance(O_T, O_{T_j}) = 0$  to both conditions.

If there are no exactly one node  $T_i$  which satisfies the requirements above we consider the search result as ambiguous and cannot return only one node as the result. It may occur when the source code was modified significantly, the target entity was changed or removed and there are 0 or 2 or more nodes in the parse tree, similar to the target node. In this case the set of all nodes is sorted according to the product of  $\overline{D_i} \cdot \overline{W}$ , where vector  $\overline{W}$  defines weights of parts of contexts.

#### D. Complexity

Wagner-Fischer algorithm [19] is used to calculate edit distances. It has a time complexity of  $O(NM)$  where N and M are lengths of two strings. Calculating edit distance of Header Contexts requires calculating edit distance between two strings at each step. For simplicity, we assume that all tokens and all header contexts have similar length. It gives a time complexity of  $O(N^2M^2)$ , where N is the length of Header contexts (in tokens) and M is length of tokens.

Calculating edit distance between two Outer Contexts has a time complexity of  $O(N^2M^2K^2)$ , where K is a length of Outer Context (depth of the parse tree).

In most cases values N, M and K are relatively small. Length of separate tokens usually ranges between 1 and 10-15, longer identifiers are rare. Header Context contains usually not more than 10-15 tokens. Outer context in case of most programming languages contains 1-3 items (e.g. a namespace and a class).

Calculating edit distance is performed for each item in set *Tree*.

Other operations have a time complexity between  $O(N)$  (calculating *NearG* and *NearL*, finding exact match) and  $O(N \log N)$  (sorting), where N is a number of items in set *Tree*.

### IV. TOOL

The tool<sup>2</sup> based on the model was designed to be easily integrated into different integrated developer environments and text editors, such as Microsoft Visual Studio and Notepad++.

#### A. Architecture

The tool is separated into 3 main parts:

- A collection of lightweight parsers and a parser generator. A parser analyzes source files written in a specific language and provides a parse tree which is then used by the core. To make development of new parsers easier a DSL-language *LightParse* was implemented along with an utility which generates lex/yacc and C# code of the parser from an input *LightParse* file.
- Core. It implements the model with algorithms. It loads and runs parsers to get a parse tree when it's necessary for saving or searching for a code fragment. A visual component with user interface ready to be integrated into different IDEs is also implemented.

- A collection of plug-ins for integrated development environments or text editors. Since the tool relies on lightweight parsers rather than on a specific IDE, and the visual part of the tool along with algorithms is provided by the core, the tool can be very easily integrated into different IDEs. A plug-in for an IDE should only display the UI component and implement simple interface, which defines 10 methods, such as getting and setting cursor position, accessing the text of currently open files and event handlers for opening and closing the IDE.

At this moment implemented lightweight parsers include: C#, Lex and Yacc, Java, XML, PascalABC.NET and a parser for our own language *LightParse*. Plug-ins for Microsoft Visual Studio, Notepad++ and PascalABC.NET [20] are developed and the tool is also integrated into a grammar editor Yacc MC.

#### B. Functionality

The tool adds a concern tree to the interface of a IDE. Concern tree may have arbitrary structure and is created by a developer. Each tree node has title and optional description and subnodes. Description length is not limited. It's displayed as a tooltip and may be edited in a separate window.

Each node may be bound to a fragment of code. In this case the node is marked with an arrow. Double click performs navigation to the code fragment if the code fragment may be identified unambiguously. Otherwise, the tool suggests several most similar code fragments. Each code fragment may be navigated to in one click and if the code fragment is found, double click updates the information in the concern tree, so next navigation will not require any additional actions.

A reverse search is also possible. The tool can find a node in the concern tree by cursor position in a current file. Along with the descriptions for tree nodes it may be used to extract some long comments from the code into the concern tree and still be able to easily find and read them.

There are several scenarios of using the concern tree. First, it may be used to maintain a "working set" of fragments, related to a current task. Concern tree is relatively small and finding the node in the tree may be much faster than finding the code fragment in one of currently open files manually.

Concern tree significantly simplifies re-creating working set when returning to a task. Instead of recalling class and method names, performing cross-reference search its only necessary to expand a subnode in the concern tree related to the task.

Concern tree is very helpful when a new developer starts working with unfamiliar project. Concern tree resembles a table of contents, it's easy to find concerns in it and each concern contains all code fragments related to it with descriptions. Reading description and navigating across the code helps to understand how the code is organized and how it works.

The functionality, concern tree examples and the tool usage scenarios were presented at CEE-SEC(R) 2015 Conference<sup>3</sup>

<sup>2</sup>Available at <https://github.com/MikhailoMMX/AspectMarkup>

<sup>3</sup><http://2015.secr.ru/lang/ru/program/submitted-presentations/aspect-markup-of-a-source-code-for-quick-navigating-a-project>

## V. CONCLUSION

We propose an approach to working with crosscutting concerns. Concerns are organized in a tree-like structure and tree nodes are bound to code fragments scattered across the project. Concern tree is added to the interface of IDE as a toolbox. Concern tree simplifies navigating across scattered fragments and is helpful for investigating and re-investigating a concern. We describe a model our approach is based on. A metrics of distance between entities in a code is defined. A description of data, stored in a concern tree is given. Algorithms of identifying a minimal amount of data to store and searching an entity in a modified source code are provided.

The model is implemented in a tool, which supports different programming languages and integrates into different editors and integrated development environments. It performs either navigation to a saved code fragment if it can be determined precisely, or shows most similar code fragments otherwise. The concern markup tool is used in development of PascalABC.NET and the tool itself.

At this moment some features of the model are not implemented yet, such as horizontal context.

We are currently collecting statistical data and enhancing algorithms to better handle most frequent changes in the source code. Some parameters, such as weights of operations need adjustments.

## REFERENCES

- [1] M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns," in *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, ser. ACoM '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 2–. [Online]. Available: <http://dx.doi.org/10.1109/ACOM.2007.4>
- [2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006. [Online]. Available: <http://dx.doi.org/10.1109/TSE.2006.116>
- [3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, ser. ECOOP '01. London, UK, UK: Springer-Verlag, 2001, pp. 327–353. [Online]. Available: <http://dl.acm.org/citation.cfm?id=646158.680006>
- [4] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The genvoca model of software-system generators," *IEEE Softw.*, vol. 11, no. 5, pp. 89–94, Sep. 1994. [Online]. Available: <http://dx.doi.org/10.1109/52.311067>
- [5] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 187–197. [Online]. Available: <http://dl.acm.org/citation.cfm?id=776816.776839>
- [6] S. Apel, C. Kastner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231. [Online]. Available: <http://dx.doi.org/10.1109/ICSE.2009.5070523>
- [7] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella, "Delta-oriented programming of software product lines," in *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, ser. SPLC'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–91. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1885639.1885647>
- [8] W. Harrison and H. Ossher, "Subject-oriented programming: A critique of pure objects," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 411–428. [Online]. Available: <http://doi.acm.org/10.1145/165854.165932>
- [9] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying, "Visual separation of concerns through multidimensional program storage," in *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, ser. AOSD '03. New York, NY, USA: ACM, 2003, pp. 188–197. [Online]. Available: <http://doi.acm.org/10.1145/643603.643623>
- [10] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeptura, and J. J. LaViola, Jr., "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 2503–2512. [Online]. Available: <http://doi.acm.org/10.1145/1753326.1753706>
- [11] S. Chiba, M. Horie, K. Kanazawa, F. Takeyama, and Y. Teramoto, "Do we really need to extend syntax for advanced modularity?" in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 95–106. [Online]. Available: <http://doi.acm.org/10.1145/2162049.2162061>
- [12] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 311–320. [Online]. Available: <http://doi.acm.org/10.1145/1368088.1368131>
- [13] M. Malevanny and S. Mikhalkovich, "Realizatsiya podderzhki aspektov programmnogo koda v integrirovannykh sredakh razrabotki[implementation of support of aspects in integrated development environments]," in *Sovremennye informatsionnye tekhnologii: tendentsii i perspektivy razvitiya: materialy konferentsii[Modern information technologies: tendencies and perspectives of evolution]*, 2015, pp. 351–353, (in Russian).
- [14] M. P. Robillard and F. Weigand-Warr, "Concernmapper: Simple view-based separation of scattered concerns," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '05. New York, NY, USA: ACM, 2005, pp. 65–69. [Online]. Available: <http://doi.acm.org/10.1145/1117696.1117710>
- [15] M. Malevanny, "Legkovesnyi parsing i ego ispol'zovanie dlya funktsii sredy razrabotki[lightweight parsing and its application in development environment]," *Informatizatsiya i svyaz'[Informatization and communication]*, vol. 3, pp. 89–94, 2015, (in Russian).
- [16] ANSI C grammar. [Online]. Available: <http://www.quut.com/c/ANSI-C-grammar-y.html>
- [17] PascalABC.NET. (in Russian). [Online]. Available: <http://pascalabc.net/>
- [18] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics – Doklady*, vol. 10, no. 8, pp. 707–710, 1965, (in Russian).
- [19] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974. [Online]. Available: <http://doi.acm.org/10.1145/321796.321811>
- [20] I. V. Bondarev, Y. V. Belyakova, and S. S. Mikhalkovich, "Sistema programmirovaniya pascalabc.net 10 let razvitiya [programming system pascalabc.net 10 years of evolution]," in *XX Nauchnaya konferentsiya Sovremennye informatsionnye tekhnologii: tendentsii i perspektivy razvitiya. Materialy konferentsii [XX Scientific conference Modern information technologies: tendencies and perspectives of evolution]*, 2013, pp. 69–71, (in Russian).

# Metric-based Approach to Anti-pattern detection in Service-oriented Software Systems

Alexander Yugov

Department of Software Engineering  
National Research University Higher School of Economics  
Moscow, Russian Federation  
yugovas@live.ru

**Abstract** — Service-based systems, as well as any other software systems, evolve over time. No matter what changes were pushing the evolution: new requirements, changing operational environment, etc., this evolution may hinder the maintenance of these systems, and thus increase the cost of their development. Permanent changes can introduce into the system some "bad" decisions – anti-patterns, which, in turn, reduce the quality of software system and require more attention of developers with support and further development. This article discusses examples of anti-patterns and methods for their automated detection. These methods will be focused on metric-based approach to analysis of service-based software systems.

**Keywords** — service-based systems, anti-patterns, specification and detection, software quality, quality of service (QoS)

## I. INTRODUCTION

Service-based style of software systems is very widely spread at the industrial development because it allows implementing flexible and scalable distributed systems at a competitive price. The result of development are autonomous, reusable, and independent units of a platform – services – that can be consumed via any network including the Internet [9].

Traditional approaches to software delivery are based on life cycle phases of the system, when in the development process became involved various teams inside a company or even by different companies [10]. Moreover, in classical approach, the focus is on one vendor supplying the entire system or subsystem. The emergence of service-oriented architecture approach introduces a model divided into levels. It enables the existence of different design approaches, whereby different parties deliver service layers as separate elements. Experience in development of joint projects, divided into separate services, shows that errors may appear in potentially dangerous areas. As part of this work, we will call these areas as anti-patterns.

Anti-patterns in software systems based on services are "bad" solutions recurring design problems. In contrast to design patterns, anti patterns are well-proven solutions that engineers should avoid. Anti-patterns can also be introduced as a consequence of various changes, such as new user requirements or operating environment changes.

This paper presents an introduction to the anti-pattern detection domain and describes proposed approach for the automated detection of anti-patterns.

## II. EXAMPLES OF ANTI-PATTERNS IN SERVICE-BASED SYSTEMS

Design (architecture) quality is vitally important for building a well thought-out, easy to maintain and evolving systems. The presence of patterns as antipattern in the system design was recognized as one of the most effective ways to express architectural problems and their solutions, and hence higher quality criterion among different systems [23].

A number of efforts have been taken to formalize the properties of the concept of "bad" practices, i.e., decisions that adversely affect the quality of the system. Despite the emerging interest to service-based systems, the literature is not really consistent with respect to pattern and anti-pattern definition and specification in this area. Indeed, the available catalogs use different classification, either based on their nature, scope or objectives.

Some completely new approaches were introduced to identify and detect code vulnerabilities and anti-patterns [22], [14]. The methods used in these campaigns were very diverse: completely manual, based on the research guidelines; metrics based on heuristic methods using rules and thresholds for various metrics; or Bayesian networks. Some approaches [1] are applicable to the application level and can be applied to initial stages of the software life cycle.

Quite a large number of methodologies and tools exist for the detection of anti-patterns, in particular, in object-oriented (OO) systems [13], [16]. However, the detection of anti-patterns in service-based systems, in contrast to the OO systems is still in its infancy. One of the last works by detecting of antipattern in service-oriented architectures (SOA) has been proposed in Moha et al. In 2012 [22].

The authors proposed an approach to the determination and detection of an extensive set of SOA anti-patterns operating such concepts as granularity, cohesion and duplication. Their instrument is able to detect the most popular SOA anti-patterns, defined in literature. In addition to these antipatterns, authors identified three antipatterns, namely: bottleneck service, service chain and data services. Bottleneck is a service that is used by many other components of the system, and as a result, is characterized by high incoming and outgoing connections affecting the response time of service. Chains of services occur when a business object is achieved by a long chain of successive calls. Data service is a service that performs

a simple operations of information search or data access, which may affect the connectivity of the component.

In 2012, Rotem-Gal-Oz [26] identified the “knot” antipattern, a small set of connected services, which, however, is closely dependent on each other. Anti-pattern, thus, may reduce the ease of use and response time.

Another example of anti-pattern is “sand pile” defined by Kr'al et al [15]. It appears when many small services use shared data, which can be accessed through the service, which represent the “data service” anti-pattern.

In the paper of Scherbakov et al. proposed “duplicate service” antipattern [5] that affects sharing services that contain similar functions, causing problems in the support process.

In 2003 Dudney et al. [8] have identified a set of anti-patterns for the J2EE applications. “Multi service” anti-pattern stands out, among others, a “tiny service” and “chatty service”. Multi service is a service that provides a variety of business operations, which have no practical similarity (for example, belong to different subsystems) that can affect service availability and response time. Tiny service is a small service with few methods, which are always used together. This can lead to the inability of reuse. Finally, an anti-pattern “chatty service” represents such services that constantly call each other, passing small amount of information.

### III. METRIC-BASED APPROACH TO THE DETECTION OF ANTI-PATTERNS

As DeMarco noted [7], in order to control the quality of development, correct quantitative methods are needed. Already in 1990 Card emphasized that metrics should be used to assess the development of software in terms of quality [3]. But what should be measured? In the above context of design rules, principles and heuristics, this question should be rephrased as follows: is it possible to express the principles of “good design” in a measurable way?

The main goal of this approach is to provide a mechanism for engineers, which will allow them to work with metrics on a more abstract level, which is conceptually much closer to real conditions of applying numerical characteristics. Mechanism defined for this purpose is called a discovery strategy:

**Detection strategy** is a quantitative expression of the rules by which specific pieces of software (architectural elements), corresponding to this rule, can be found in the source code.

By this reason, the detection strategy is a common approach to analysis of the source code model using metrics. It should be noted that in the context of the above definition, “quantitative expression of the rule” means that the rule should be properly expressible using metrics. The use of metrics in detection strategies grounded filtering mechanisms and composition. In the following subsections, these two mechanisms will be considered more detailed.

The key problem in data filtering is reducing the initial collection of information, so that there remain only those values that are of particular value. This is commonly referred to as data reduction [12]. The aim is to detect those elements of

the system, which have special properties. Limits (boundaries) of the subset are determined on the basis of the type of filter. In the context of the measurement process with respect to the software, we usually try to find the extreme (abnormal) values or those values that lay within a certain range. Therefore, distinguish types of filters [19]:

- Marginal filter is a data filter, in which one limit (border) in the result set is clearly identified with a corresponding restriction of the original data set.
- Interval filter is a data filter, in which the lower and upper limits of the resulting subset are explicitly specified in the definition of the data set.

Marginal filters consist of two depending on how we specify the borders, resulting dataset limiting filters may be semantical or statistical.

- Semantical. For these filters two parameters must be specified: a threshold value that indicates a limit value (to be explicitly indicated); and the direction that determines whether the threshold upper or lower limit of the filtered data set. This category of filters is called semantical as the choice of options is based on the semantics of specific metrics in the framework of the model chosen for the interpretation of this metric.
- Statistical. Unlike semantical filters, statistical ones do not require explicit specifications for the threshold, as it is defined directly from the original data set using statistical methods (e.g., scatter plot). However, the direction is still to be specified. Statistical filters are based on the assumption that all the measured entities of the system are designed using the same style, and therefore, the measurement results are comparable.

In this paper, a set of specific data filters of the two previous categories were used. Basing on practical use and interpretation of the selected models, these filters may be grouped as follows:

- Absolute semantic filters: *HigherThan* and *LowerThan*. These filtering mechanisms are parameterized by a numerical value representing the border. We will only use data filters are to express “clear” design rules or heuristics, such as “class should not be associated with more than 6 other classes.” It should be noted that the threshold is specified as a parameter of the filter, while the two possible directions are defining by two particular filters.
- Relative semantic filters: *TopValues* and *BottomValues*. These filters differentiate the filtered data set according to the parameter that determines the number of objects to be recovered, and do not indicate the value of the maximum (or minimum) values are permitted in the result set. Thus, the values in the result set will be considered with respect to the original data set. The parameters used may be absolute (for example, “select 20 objects with the highest values”) or percentile (for example, “to remove 10% of the measured objects with the lowest values”). This type of filter is useful in situations where it is necessary to consider the highest

(or lowest) values of a given data set, rather than indicating the exact thresholds.

- **Statistics: scatter plots.** Scatter diagram is a statistical method that can be used to detect outliers in the data set [11]. Data filters based on these statistical techniques, which, of course, not limited to only the scatter diagrams, are useful in the quantification of rules. Again, we need to specify the direction of the deviation of adjacent values based on design rules of semantics.
- **Interval Filters.** Obviously, for the data interval it is necessary to define two thresholds. However, in the context of the detection strategies, where, in addition to the mechanism of filtering, the composition mechanism exists, filter interval is defined by two composition of two semantic absolute filters of opposite directions.

Unlike simple metrics and interpretation models of it, detection strategy should be able to draw conclusions on the basis of a number of rules. Consequently, in addition to the filtering mechanism, which supports the interpretation of the particular metric results, we need a second mechanism for comparing the results of calculations of a number of metrics – a mechanism of composition. Composition mechanism is a rule combining the results of calculating several metric values. In the literature three composition operators were observed: “and”, “or” and “butnot” [19].

These operators can be discussed from two different perspectives:

- From a logical point of view. These three operators are a reflection of rules to combine multiple detection strategies, where operands are descriptions of the design characteristics (symptoms). They facilitate reading and understanding of the detection strategy, because operators of composition are generally expressed in the form of quantitative characteristics, so it is similar to the original wording of the informal thoughts. From this point of view, for example, the operator «and» presupposes that the investigated object has both symptoms that are combined by the operator.
- From the point of sets. This view helps to understand how to build the ultimate result of the detection strategy. The initial set of calculation results on each of the metrics is carried out through the filtering mechanism. Then remains limited set of system elements (and calculated metrics for these elements), which are interesting for further investigation. The resultant plurality of filtered sets should be merged with the operators using the formulation. Thus, in terms of operations on sets, the operator “and” will correspond to the operation of intersection ( $\cap$ ), the operator “or” to reunion operation, and the operator “butnot” to minus operation.

#### IV. DEFINITION OF DETECTION STRATEGY

This section will be written in the formation of a strategy on the example of the detection of a particular anti-pattern “God Object” [25]. The starting point is the presence of one (or

more) of the informal rules that describes the problem situation. In this example, we will proceed from the three heuristics found in the book of Riel [25]:

- The top-level services should share equally the responsibility.
- Services should not contain large amounts of semantically separate functions.
- Services should not have access to fields or properties of other services.

##### A. Step-by-step Strategy

The initial step to create a detection strategy is to translate the set of informal rules into symptoms that can be evaluated by a particular metric. In the case of God Object anti-pattern, the first rule refers to an equal sharing of responsibilities among services, and therefore it refers to service complexity. The second rule tells us about the intensity of communications among this service and all other services; thus, it refers to the low cohesion of services. The third heuristic describes a special coupling i.e., the direct access to data items manipulated by other services. In this case, the symptom is access to “foreign” data.

The second step is to find appropriate metrics, which evaluate more precisely every of the discovered properties. For the God Service anti-pattern, these properties are complexity of the service, cohesion of the service and access to data from other services. Therefore, we found the following set of metrics:

- **Weighted Method Count (WMC)** is the sum of the static complexity of all methods in a class [6]. We considered the McCabe’s approach as a complexity measure [20].
- **Tight Class Cohesion (TCC)** is the relative number of directly connected methods [2].
- **Access to Foreign Data (ATFD)** represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [18].

The next step is to select an appropriate filtering scheme that should be applied to all metrics. This step is mainly done basing on the rules described earlier. Therefore, as the first symptom is a “high service complexity” the *TopValues* relative semantical filter was chosen for the WMC metric. For the “low cohesion” symptom it was also chosen a relative semantical filter, but now the *BottomValues* one. For the third symptom, an absolute filter was selected as we need to catch any try to access a “foreign” data; thus, we the *HigherThan* filter will be used.

One of vital issues in creating a detection strategy is to choose proper parameters (i.e., threshold values) for all data filters. Several approaches exist to do this, but now we just take a 25% value for both the *TopValues* filter for to the WMC metric and to the *BottomValues* filter for the TCC metric. As for filter boundary for the ATFD metric, the decision is pretty simple: no direct access to the data of other services should be allowed, therefore, the threshold value is 1.

The final step is to join all the symptoms, with applying of the special operators described before. From the unstructured heuristics as presented in [25], it was inferred that all three symptoms should be combined if a service is supposed to be a behavioral God Object.

The intention of this work is to use detection strategies in rule definitions in order to facilitate detection of anti-patterns in service-based software systems i.e., to select such areas of the system (subsystem) that are participated in a particular anti-pattern. From this point of view it should be emphasized that the detection strategy approach and the whole method is not limited by finding problems, but it also can facilitate completely different objectives too. For instance, different investigation purposes could be in reverse engineering [4], design pattern detection [18], identification of components in legacy systems [27], etc.

## V. IMPLEMENTING A TOOL FOR DETECTION OF ANTI-PATTERNS IN SERVICE-BASED SYSTEMS

### A. Description of Metrics

Calculations intended to detect antipatterns is conducted basing on several basic metrics:

- incoming call rate;
- outgoing call rate;
- response time;
- number of service connections;
- cohesion with other services;
- etc.

Each metric has its specific model and its specific algorithm to calculate. Values of this metric have decisive influence on detection of services participating in antipatterns.

In calculation of metrics, objective measures of occurrence pattern interestingness of data mining like confidence and support are used. These are based on the structure of discovered patterns and the statistics underlying them.

A measure for association rules of the form  $X \rightarrow Y$  is called support, representing the percentage of transactions from a log database that the given rule satisfies. This is intended to be the probability  $P(X \cup Y)$ , where  $X \cup Y$  indicates that a transaction contains both  $X$  and  $Y$ , that is, the union of item sets  $X$  and  $Y$ .

Another objective measure for association rules from data mining is confidence, which addresses the degree of certainty of the detected association. In classical data mining this is taken to be the conditional probability  $P(X \cap Y)$ , that is, the probability that a transaction containing  $X$  also contains  $Y$ . More formally, confidence and support are defined as

$$\begin{aligned} \text{Support}(X \rightarrow Y) &= P(X \cup Y), \\ \text{Confidence}(X \rightarrow Y) &= P(X \cap Y). \end{aligned}$$

In general, each measure of interestingness is associated with a threshold, which may be controlled. For calculation of

metrics each final value of metric is confidence (which is calculated not as in classical data mining but more complexly) divided by support measure (which is calculated in the same manner as in classical data mining).

Further, each metric is described in more details.

Incoming and Outcoming Call Rates. The model for calculation of IncomingCallRate metric is call matrix. This matrix represents calls services make to each other. For building this matrix and some other models, we need to identify the order of calls. This information does not stored in logs, therefore, the first task is to mine service calls from log. Procedure of mining calls consists of several main steps. The first is ordering log events by traces. This is necessary because occurrence of events in particular order in boundaries of one trace gives us evidence of one particular service call. To mine all the service calls properly it is needed to sort events in the log chronologically within every trace. Once ordering on both levels (trace and timestamp) is finished, we can go through the log and reconstruct service calls. Received values in mined matrix will represent generalized number of calls among services for as IncomingCallRate as OutcomingCallRate.

Response Time. Response time metric represent general bandwidth of a particular service. This parameter is crucial for systems having high load. Calculation of this metric uses assumptions made for both metrics IncomingCallRate and OutcomingCallRate but with some modifications. As we are aimed here at measure of time characteristic, the object to explore will be time stamp parameter of the log. Given defined algorithm for incoming and outgoing call rates, we modify it with calculation of time prospect. Instead of just number of calls, we calculate general length of service response. In such a way the summarized time while service was busy is calculated. As a result of precious calculations, the matrix of general time every service spent on work was obtained. Following step is to normalize real values, i.e. to measure not in absolute number but in relative number. This relative number will show percentage of time where the service was working on processing calls. This metric can be used for detection of both highly loaded services and rarely used services.

Cohesion with Other Services. For calculation of this metric classical data mining rules are implemented. For this the conditional probability  $P(X \cap Y)$  is taken. That is, the probability that a transaction containing  $X$  also contains  $Y$ . Additionally, the special rule for ordering is added. This means that  $X \rightarrow Y$  and  $Y \rightarrow X$  is different relations. I.e. we observe not only occurrence at one trace but also the order of occurrences. High rate of confidence of this metric is evaluated as high cohesion of several services and, therefore, high behavioral dependency.

Number of Service Connections. All the previous metrics were dynamic characteristics of a system under consideration while number of service connections is a static property of the system. For mining this property the incidence matrix of service calls is enough. If one service called once another service, we do not consider the same connections in future. Obtained incidence matrix allows us to calculate all existing connections in the system.



The basic model to calculate each of metrics is Graph model (fig. 1) which is extended in each particular metric calculation algorithm with specific attributes. As part of this work, it is assumed that each object, once appeared in the system, initiates a sequence of operations to be performed on the object. This sequence of operations is called workflow. It is worth noting that not every service-oriented system is based on this principle, but we will consider only such systems.

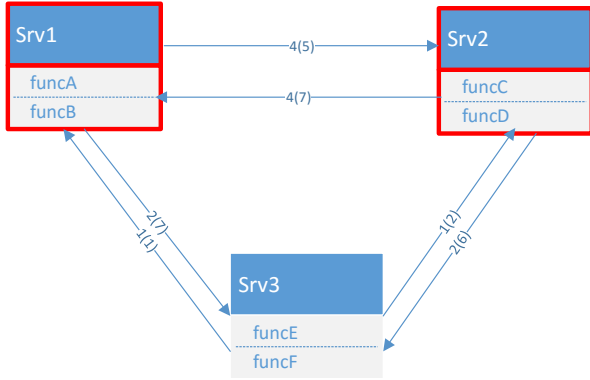


Fig. 1. Base graph model for calculation of metrics.

In this model, services of a software system are presented by graph nodes. Arcs of the graph represent the call ratio, i.e., oriented arc from Srv1 to Srv2 shows that Srv1 in the process of operation calls one of Srv2 functions. Depending on what metric should be calculated, edges of the graph are marked by specific values. For example, on fig. 1 arcs are labeled by amount of calls in a particular direction and, in parentheses, some weighted value of the transmitted data.

#### B. Extracing Data from Event Logs

The main weakness of previously observed works was the necessity to modify source code of a particular system in order to evaluate concrete metrics. In this work we use event logs to create a process model of the system and calculate metrics basing on this model. To apply these, it is assumed that the information system records data of events. These logs also contain unstructured and irrelevant data, e.g. information on hardware components, errors and recovery information, and system internal temporary variables. Therefore, extraction of data from log files is a non-trivial task and a necessary pre-processing step for analysis. Business processes and their executions related data are extracted from these log files. Such data are called process trace data. For example typical process trace data would include process instance id, activity name, activity originator, time stamps, and data about involved elements. Extracted data are converted into the required format.

To be able to analyze log content, the log should have specified structure. In our case the minimal requirements for log is as follows:

- *TraceID*: shows the identity for a particular trace;
- *ServiceID*: shows the identity for a particular service;
- *FunctionID*: shows the identity attribute for a particular function in the service;

- *Timestamp*: shows the time of occurrence of the event.

The log sample is presented in table 1.

TABLE I. SOURCE LOG SAMPLE

TraceId	Service	Function	TimeStamp
1	Srv2	C	2015-06-15 00:25:20
1	Srv1	A	2015-06-15 00:33:24
2	Srv4	F	2015-06-15 00:32:25
3	Srv3	E	2015-06-15 00:24:13
1	Srv2	C	2015-06-15 00:31:52
3	Srv1	B	2015-06-15 00:34:05
4	Srv4	G	2015-06-15 00:25:12
3	Srv3	E	2015-06-15 00:26:28
4	Srv1	A	2015-06-15 00:28:21
4	Srv2	C	2015-06-15 00:30:32
2	Srv1	A	2015-06-15 00:29:48
2	Srv2	C	2015-06-15 00:29:51

Each field included in log has its own purpose in future usage. *TraceID* is needed for distinguishing events among execution sequences, i.e. for majority of metrics it is necessary to connect events in boundaries of one trace. Moreover, inside traces events appears in chronological order. That is why timestamp is included in log format.

*ServiceID* and *FunctionID* describe source of each event. In addition, dimensions of functions and services are main structural units in analysis and creation of models.

#### C. Specification of Rule Cards

The rule cards is storing in XML format. The structure of XML represents scheme of rule card structure. The scheme of XML is presented in fig. 2 in graphical mode. In fig. 3 for more detailed view in XSD standard. The XML should have specialized namespace: "RuleCardNS". The root element is "RuleCard". It has name element called "AntipatternName". This also plays the role of identification attribute.

Each Rule is defined through type attribute, metric value and its own name. The type attribute describes what metric (from a set of available metrics) should be calculated. Metric value refers to specific value of calculated metric, which shows whether the service under analysis has a particular symptom or not. Finally, rule name is an identification property for rule.

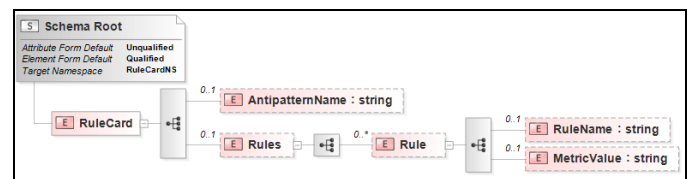


Fig. 2. Structure of antipattern XML

```

<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="unqualified"
elementFormDefault="qualified"
targetNamespace="RuleCardNS">
  <xsd:element name="RuleCard">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="AntipatternName" type="xsd:string" />
        <xsd:element name="Rules">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element maxOccurs="unbounded" minOccurs="1"
name="Rule">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="RuleName" type="xsd:string" />
                    <xsd:element name="MetricValue" type="xsd:string" />
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xs:schema>

```

Fig. 3. Detailed structure of antipattern XML.

#### D. Description of Research Prototype of Analytic System

To automate process of anti-pattern detection the research prototype of information system, which implements the described approach, has been developed. The scheme of the proposed approach is shown in fig. 4. The workflow of the software system consists of several steps. At the point of entry, the program takes log, which is reading from the relational database implemented in SQL Server, and rule card describing rules to detect particular antipattern.

General workflow structure is presented in fig. 5. It starts with reading input data, which are:

- log from some software system implemented according to SOA principles;
- rule card describing all the rules and metrics needed for detection of each particular antipattern.

Once the XML with antipattern description is read the system starts calculation of metrics. Each metric is calculated against its specific algorithm. So for each rule the process of metric calculation has been launching. First, the special model used for analysis of a particular metric is build. All the models were defined previously. Then with use of received model metrics are calculated. As a result of this process, services suspected in participation in the antipattern are selected.

Next step is to integrate results received in threads of calculation of metrics. The integration is conducted as intersection of result sets from previous threads. Finally, we obtain set of suspicious services, which are parts of antipattern. Commonly there are several services, but is always can be that

just one service represents antipatterns or no such services at all were discovered.

Results of analysis is depicting in general graph representation (fig. 4). Nodes in this graph are services and edges in this graph are direct references among services.

Each node represents one service observed in the system whose log has been observed. As for example on fig. 5, nodes such as for services Srv2, Srv3, and Srv4 represent proper developed services, i.e. they are not participated in antipatterns. Suspicious services are marked with “!” sign, that means that this particular service is a part (or is whole) of antipattern. In our example this is service number 1 (Srv1).

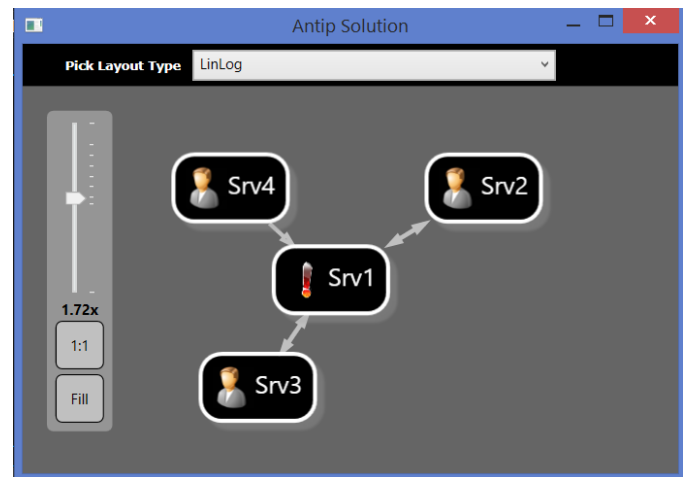


Fig. 4. Graphical representation of results.

Edges represent calls made of one service to another one. Concerning example from fig. 2.6, Srv4 calls Srv1 therefore one edge directed from Srv4 to Srv1 is depicted. Srv1 and Srv2 calls functions of each other therefore the edge is bidirectional.

## VI. CONCLUSION

This work addresses the issue of necessity of monitoring circumstance of software systems implemented through service-based approach in conditions of continuous development and enhancement when number and complexity of systems is expanding faster than a human being can handle.

During the exploration of process mining and data mining domains the general service-based specific antipattern detection rules were invented. All rules consist of several metrics and its specific values, describing symptoms of antipatterns. At the time, five metrics are available for usage in detection rules: incoming call rate, outgoing call rate, response time, cohesion, and number of service connections. With applying these metrics several antipatterns was specified and algorithms for it detection were introduced.

Algorithms of antipattern detection based on metric calculation were implemented as a software tool (research prototype), which allows by specifying rule cards in XML format and log in SQL Server database to detect antipatterns. The software tool is developed with usage of Windows Presentation Foundation framework.

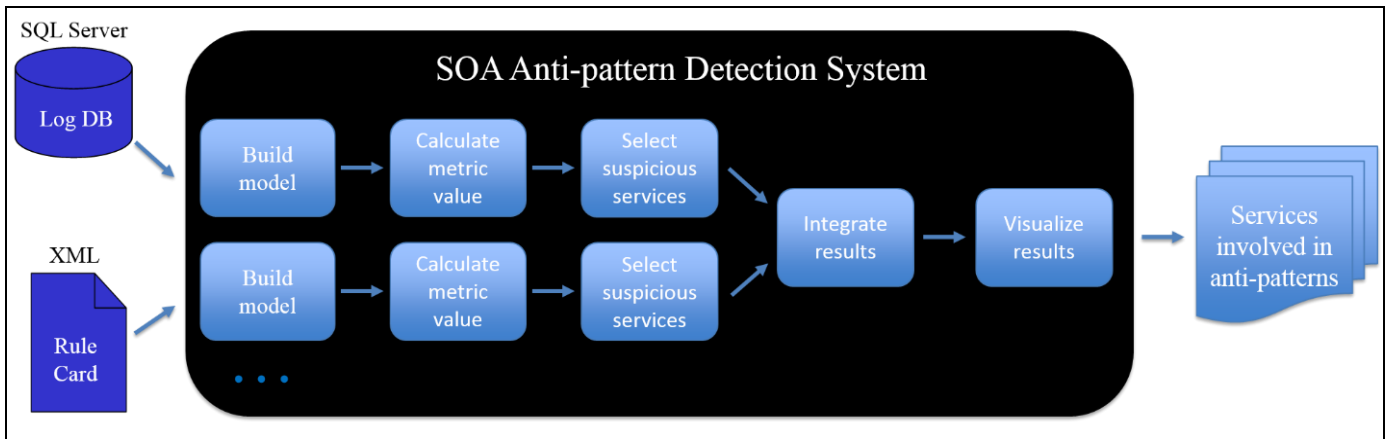


Fig. 5. Base graph model for calculation of metrics.

It is planned that in future the information system will be refined according to analysis of real life logs from and number of available metrics and possible to detect antipatterns will be significantly greater. The following step will be introducing dynamic analysis of system behavior in addition to implemented analysis of static footprints. Furthermore, some fuzziness can be introduced for the evaluation of the threshold values thus to make antipattern detection rules more flexible.

#### REFERENCES

- [1] D. Arcelli, V. Cortellessa, C. Trubiani. Experimenting the Influence of Numerical Thresholds on Model-based Detection and Refactoring of Performance Antipatterns. ECEASST 59 (2013).
- [2] J.M. Bieman and B.K. Kang. Cohesion and Reuse in an Object-Oriented System. Proc. ACM Symposium on Software Reusability, apr 1995.
- [3] D. Card and R. Glass. Measure Software Design Quality. Prentice-Hall, NJ, 1990.
- [4] E. Casais. State-of-the-art in Re-engineering Methods. achievement report SOAMET-A1.3.1, FAMOOS, October 1996.
- [5] L. Cherbakov, M. Ibrahim, and J. Ang, "Soa antipatterns: the obstacles to the adoption and successful re-alization of service-oriented architecture".
- [6] S. R. Chidamber and C. F. Kemerer. A Metric Suite for Object-Oriented Design. IEEE Transactions on Software Engineering, 20(6):476–493, June 1994.
- [7] T. DeMarco. Controlling Software Projects: Management, Measurement and Estimation. Yourdan Press, New Jersey, 1982.
- [8] B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, J2EE Antipatterns, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2002
- [9] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, August 2005.
- [10] G. Farrow. SOA antipatterns: When the SOA paradigm breaks // IBM Developer Works [Online]. Available: [http://www.ibm.com/developerworks/library/wa-soa\\_antipattern/](http://www.ibm.com/developerworks/library/wa-soa_antipattern/)
- [11] N. Fenton and S.L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, second edition, 1997.
- [12] P.G. Hoel. Introduction to Mathematical Statistics. Wiley, 1954.
- [13] M. Kessentini, S. Vaucher, and H. Sahraoui. "Deviance From Perfection is a Better Criterion Than Closeness To Evil When Identifying Risky Code" in Proceedings of the IEEE/ACM ASE. ACM, 2010, pp. 113–122.
- [14] F. Khomh, M. D. Penta, Y.-G. Güehéneuc, G. Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering 17(3):243–275, 2012.
- [15] J. Kr'al and M. Zemlicka, "The most important service-oriented antipatterns," in ICSEA, 2007, p. 29.
- [16] M. Lanza and R. Marinescu, Object-Oriented Metrics in Practice. Springer-Verlag, 2006.
- [17] M. Lorenz and J. Kidd. Object-Oriented Software Metrics. Prentice-Hall Object-Oriented Series, Englewood Cliffs, NY, 1994.
- [18] R. Marinescu. Detecting Design Flaws via Metrics in Object-Oriented Systems. In Proceedings of TOOLS USA 2001, pages 103–116. IEEE Computer Society, 2001.
- [19] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04). Los Alamitos CA: IEEE Computer Society Press, 2004, pp. 350–359.
- [20] T.J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, 2(4):308–320, dec 1976.
- [21] P. Mihancea. Optimization of Automatic Detection of Design Flaws in Object-Oriented Systems. Diploma Thesis, "Politehnica" University Timisoara, 2003.
- [22] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Güehéneuc, B. Baudry, J.-M. Jezequel. Specification and Detection of SOA Antipatterns. In International Conference on Service-Oriented Computing (ICSOC). Pp. 1–16. 2012
- [23] M. Nayrolles; N. Moha; P. Valtchev. Improving SOA Antipatterns Detection in Service Based Systems by Mining Execution Traces in Proceedings of the 20th Working Conference on Reverse Engineering (WCRE'13), pp. 321–330, IEEE, 2013.
- [24] D. Ratiu, S. Ducasse, T. Girba, and R. Marinescu. Using History Information to Improve Design Flaws Detection. In Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004). IEEE Computer Society, 2004.
- [25] A.J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, 1996.
- [26] A. Rotem-Gal-Oz, SOA Patterns, 1st ed. Manning Publications, 2012.
- [27] A. Trifu. Using Cluster Analysis in the Architecture Recovery of OO Legacy Systems. Diploma Thesis, Karlsruhe and the "Politehnica" University Timisoara, 2001

# Technology for application family creation based on domain analysis

Gudoshnikova Anna

Chair of informatics

St.-Petersburg State University

St.Petersburg, Russia

Email: gudoshnikova.anna@gmail.com

Yurii Litvinov

Software Engineering chair

St.-Petersburg State University

St.Petersburg, Russia

Email: y.litvinov@spbu.ru

**Abstract**—The theme of code reuse in software development is still important. Sometimes it is hard to find out what exactly we need to reuse in isolation of context. However, there is an opportunity to narrow the context problem, if applications in one given domain are considered. Hence, the problem of domain analysis arises. On the other hand, there is metaCASE-technology that allows to generate code of an application using diagrams. The main objective of this article is to present the technology for application family creation which connects the metaCASE-technology and domain analysis. We propose to use feature diagrams to describe variability in a domain and then create domain-specific visual language that allows to connect and configure existing feature implementations thus producing an application. This technology supposed to be especially useful for software product lines.

**Index Terms**—domain analysis; metaCASE-technology; domain-specific language; application family

## I. INTRODUCTION

The term “reuse” in software engineering is closely associated with context. Reuse objects can be programs, parts of programs, specifications, requirements, architectures, test plans, etc. Reuse of one object leads to reuse of another object. This means, there is a need to reuse something more than just code, i.e. there is a call for increasing the abstraction level. It is commonly supposed that reuse, as some kind of activity, can be divided into groups according to what should be reused: components, process for gaining the product, technology or knowledge. At all accounts any reuse object cannot be discussed without environment, where the given object exists. Hence, the context problem still remains.

However, if we reuse objects in one domain, the context issue may be narrowed. The product line implies that there is a common part, it can be: (1) architecture, (2) components, (3) algorithms, (4) methods, etc. — and this part exists in the same context. This fact facilitates the reuse problem. Consequently, the common part must be reused.

Gathering information about the domain is the crucial step in the whole process of software development. Nowadays applications in one domain are often designed

independently; this approach leads to increase of development time and cost. Usually such applications have similar functionality, so the reuse problem moves to the forefront in an attempt to speed up the development and to decrease the cost for systems in one domain. The reuse process in one domain supposes the necessity of the domain analysis activity. At present domain analysis in software life cycle is performed in informal way. There are some domain analysis tools, but such tools are not integrated with development tools. As the result of the domain analysis activity some diagrams just are put up on the board, and do not take part in following process of software design. The risk of incorrect understanding of domain-dependent knowledge increases. Therefore, many peculiarities of the domain may be missed in development process because of the factor of human error. This fact may lead to development of the product which does not satisfy requirements at all. Hence, there emerged a need for a tool in which domain analysis activity would play a vital role in software development process, i.e. based on this activity would be possible to generate some design model, so developers and other process actors could rely on this model. At the present day there is no tool that could allow to solve this problem.

One possible solution for this problem is the use of domain analysis tool in model-driven development, or, more precisely, domain-specific modelling. Domain-specific approach uses visual languages to specify system under development, but, contrary to general model-driven approach which uses general-purpose visual languages like UML, domain-specific languages are tailored specifically for given domain or a set of problems. Existing studies [1], [2], [3], [4] show that due to closeness to a problem domain and the ability to generate complete application by visual models domain-specific languages boost development productivity by 3 to 10 times compared to general-purpose languages. It is clear that developing a tool for domain-specific language “from scratch” for each domain will be prohibitively costly, so special systems are used that allow to declaratively specify syntax of a language and to automatically generate such tools as visual editor, source code generators, constraints checkers and so on. Such



systems are called DSM platforms, most known of these is MetaEdit+ [5], [6], [7], Eclipse GMP [8], [9], Microsoft Modeling SDK [10].

Main idea of domain-specific modeling is to use a number of visual languages in one tool to develop a complete system. Every language can provide a different point of view on a system. We propose to exploit this idea to automatically produce useful artefacts from the results of domain analysis thus seamlessly integrating this phase into development process (such as [11]). For that, we will use specific visual language to perform domain analysis and to build domain model, language simple enough to be useful to analysts and domain experts who do not necessarily possess programming skills. Then, using this domain model, we will generate actual domain-specific language that will allow to configure various existing pre-built components and integrate them to generate a working application. As we will see, this language will also typically be very simple so that non-programmers can use it. The only real coding in the proposed approach occurs when creating components from which applications will be built, but for product lines these components will already exist anyway, as they will in a case when a team develops many applications in one domain for some time. Not all steps in proposed approach are fully automatic, as a visual language needs tailoring after generation from domain model — we still need to manually specify shapes of its elements (to be familiar for domain experts) and configure properties which depend on existing components and can not be derived from domain model. It is also possible that generated application will need tailoring by hand, but generation can significantly lower the effort needed to create application.

Main contribution of this research-in-progress paper is a novel approach to product line development and assets reuse. Also an implementation of technology which uses this approach is presented. Our technology is based on QReal DSM platform [12], an open source tool developed by Software Engineering chair of St. Petersburg State University<sup>1</sup>. An evaluation of proposed approach is also presented, but on a rather simple problem, so a much wider evaluation is needed for this study to be considered complete, such as the applicability of this approach to complex real-life situations and determining actual productivity boost on real-life problems.

The rest of this paper is structured as follows: in section II most important terminology for domain analysis is given, also related works are considered. In section III we present our method and its implementation as development platform, in section IV an example of application of our approach is given, we will consider a family of Android gamepads for remote control of various robot models. Section V concludes the paper.

<sup>1</sup>GitHub repository and home page of QReal project, URL: <https://github.com/qreal/qreal> (03.04.2016)

## II. DOMAIN ANALYSIS APPROACHES

There is no any clear and long-standing definition of the term “domain analysis”. Almost all papers, in which this term is considered, go back to 80s-90s of the twentieth century. It was then that scientists, taking into account rapidly growing technologies, were thinking about global reuse. Always projects are developing for concrete user needs, so then the term “domain” took the definition. Domain is the field of expertise, problems in which the software intends to solve. According to Rugaber [13], the domain is described in terms of glossary, some assumptions, architecture approach and literature.

Then the question arise, how we need to analyze the domain for acquiring the necessary information. At present, the information gathering into knowledge bases is understood under the term “domain analysis”. Although, Prieto-Diaz [14] confirms that domain analysis is an activity, which is held before system analysis and its output is used for system analysis to the same degree as system analysis's output is used for system design. There are other definitions of the term “domain analysis”. Ferre [15] has presented definitions, such as: (1) the process of identification, organization and presenting the relevant information of a given domain, (2) the process, in which the customer's knowledge are identified, concretized and systemized. The relevant information of the domain should be presented in objective, readily available way, such way is called “domain model”. Mernik [16] specifies that the domain model includes not only glossary, but also must describe commonalities and variabilities of terms. Such model should precisely set bounds of the domain, i.e. clear and exact characterize a range of questions, which are considered in the domain. Term variabilities allow to define exactly, what information must be specified in concrete system implementation. Term commonalities are used for defining a set of shared operations between different applications. Implementing commonalities and adding the gained model with information, which can be specified in instance of the concrete system, a set of different systems can be obtained based on one common model. In such manner, based on one domain model, the set of different systems in given domain can be implemented. Taking into account definitions above mentioned, we can conclude that domain analysis is the activity of forward system analysis, which goal is to provide the domain model.

As stated above, at present in many software companies the term “domain analysis” is understood as information gathering into some knowledge bases, but it is obvious that there are disadvantages of this approach. It may lead to incomplete glossary, absence of agreements about understanding some terms in the domain, so any misunderstanding of domain can result in an improper product. Therefore, several dozens of years ago were introduced some formal approaches for domain analysis. Here will be mentioned some of them. The main objective of any

domain analysis approach is to produce the domain model.

Despite different understanding of the term “domain analysis”, Arango [17] showed that all formal domain analysis methods follow the general process for obtaining the domain model. This process includes next stages: (1) domain characterization, (2) data collection, (3) data analysis, (4) classification and finally (5) evaluation of domain model. There are following domain analysis approaches: 1) DARE (Domain Analysis and Reuse Environment) [18]. The crucial idea of this method is to create the domain book, that will include the universal architecture and library of reusable components. 2) DSSA (Domain-Specific Software Architectures) [19]. Given approach allows to create a domain glossary with the aid of use case analysis. 3) ODE (Ontology-based Domain Engineering) [20]. This approach connects the ontology idea with object-oriented approach. Ontology includes terms and their connections, definitions, properties and constraints. Library of objects is built based on mapping ontology with object-oriented entities. 4) FODA (Feature-Oriented Domain Analysis) [21]. This method has got popularity among scientists in the research area because of its simplicity for non-programmers. The main idea of this approach is creating feature model. This model describes functionality, which the future product should possess. Such model must note what features are compulsory for implement in any instance of application in a given domain, what features must be implemented but there is some alternative between them, and present features, which may be implemented but not compulsory. This model can be easily built by expert in the domain.

Concerning product line creating with the aid of using domain model, Estublier [22] presented approach which is based on some aspects and requirements. These entities were proposed by authors. Such approach based on MDE methodology. Domain model is considered as metamodel, which is described on MOF or UML. There is an interpreter, which translates each term in metamodel into Java class, and concrete models — into instances of these classes. Domain model is accompanied with feature model, which include some external behavior of the system. Authors use aspect-oriented techniques for feature implementing and following their mapping with terms in domain model. Consequently, there is a close interaction between domain modeling and feature modeling. It seems that such approach is a bit complicated for non-programmers. In addition, there is no any industrial use of this method, but it is worth noting that authors describe appliance in this article [23].

### III. PROPOSED APPROACH

In our approach we will use some ideas of Feature-Oriented Domain Analysis (FODA) method to perform domain analysis and to create feature models. For this we will use visual editor that implements feature diagrams and is easy enough for domain experts. Then, when feature

models are ready, each feature is implemented as reusable and configurable component on selected implementation language (C#, C++, Java and so on) and feature library is formed as a collection of such components. This process requires qualified programmers and requires more effort than to simply create one application, but it allows to reuse features from feature library to create as many applications as needed. Also, this process is scalable, so we may add new features into feature library later, thus allowing to create more complex applications. At this stage of development domain experts shall work with programmers, and they shall use feature diagrams as an input for creation of feature library to simplify matching between features and components in feature library.

Next step is to create domain-specific language that allows to combine and configure features from feature library to implement applications in given domain. This is where our approach differs from common reuse strategies. Naive approach would be to generate an application directly from feature diagram, somehow marking features that shall be included into application, and it actually works fine when domain variability is low [24]. But more common is the situation when features themselves have properties that allow to configure them, those properties can have different types. Also, components may be related to each other in different ways, be used in configuration of one another, or some of their properties may be meaningless in absence or presence of other feature. Those rules may be implemented implicitly in application generator and require that programmers will always observe them, but we propose that these rules will be captured explicitly by dedicated domain-specific language. Such language may make models that do not observe those rules syntactically incorrect, and it will greatly reduce the possibility of human error and reduce knowledge required to efficiently use programming system.

By using DSM platforms one can relatively quickly create domain-specific language that will capture domain knowledge, but we already have feature diagram, so we actually can generate the language using it. Generator takes feature diagram as input and produces metamodel of a language. Metamodel is a visual model of a language syntax, that can be opened and edited in yet another visual editor that is part of DSM platform, this editor is called metaeditor. Features from feature diagram become entities in metamodel, this metamodel is then edited to provide shape and a list of properties for each entity. Any vector image can play the role of shape, so the best practise is to select shape that is similar to a feature it depicts. For example, if an application can have buttons, “button” becomes entity in domain-specific language and looks like a button on a diagram. The same happens with properties — for each feature they are added in metaeditor to corresponding language entity with respect to feature library that actually implements this feature and uses the property to configure it. Properties have name, type and



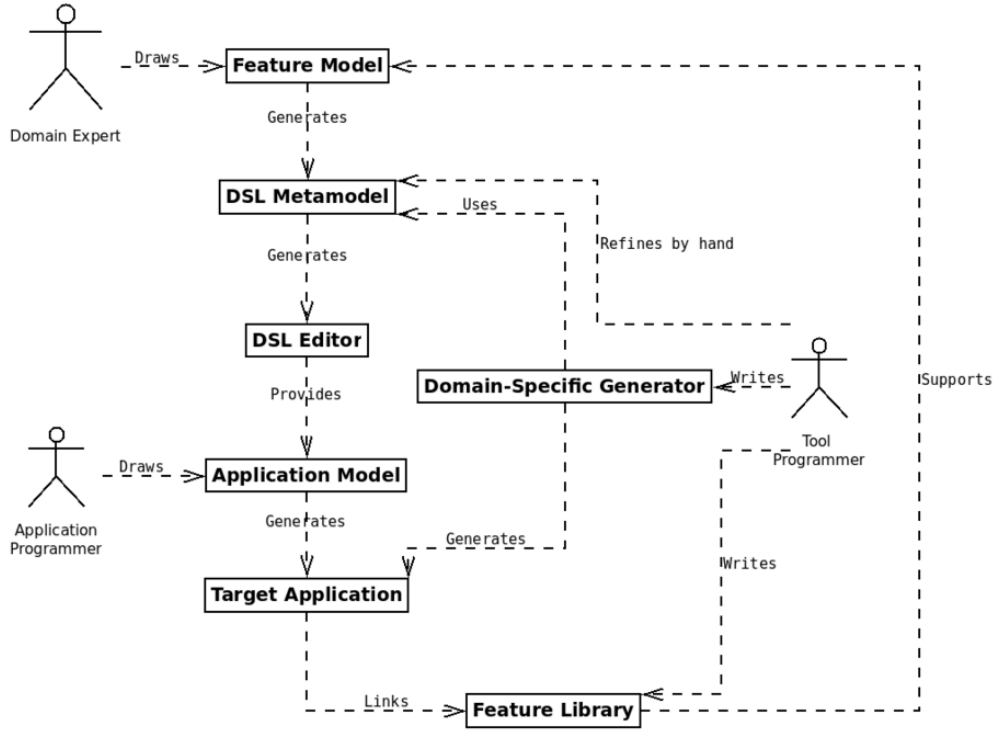


Figure 1: Relations between artefacts and roles in proposed approach to domain components reuse.

default value. On this step it is also possible to define some constraints on a metamodel that will be checked when model will be edited. If some constraints are violated, user will immediately receive warning, which makes errors in a target application even less likely to occur.

On a next step we use editor generator of the DSM platform to create visual editor for our newly created language. This step is fully automated, and when an editor is generated and loaded into DSM platform, we can use it to create diagrams that specify target applications.

The next thing we need is to generate actual code on target textual language that will call feature library and glue features together. For this we shall return to metamodel level and define generation rules for metamodel. This step is performed only once for a given domain after the feature library and metamodel are finished, and then the same generator is used for each application created by using of the technology. Recommendations for development of domain-specific generator are well-known in DSM literature (for example, [7]): it is the best to write first application by hand, then draw a model that is supposed to be generated into this application, then find the places in handwritten application that shall be parameterised by information from model and let the generator replace such handwritten parts with data from model. This process is continued until handwritten application becomes a template that is filled by generator with information taken from model. Handwritten application and, consequently, a

generator shall extensively use feature library to minimize the amount of code that is generated directly, in ideal case generator shall produce merely a glue code that binds components from feature library together.

After all steps above are finished we have feature library, visual editor for simple domain-specific language that allows to describe how features are combined and configured in a concrete application, and a generator that automatically produces complete application by a model in domain-specific language using feature library as domain-specific runtime [7]. Now we may create as many applications as we wish by just drawing models and automatically generate complete executable code. Theoretically. Of course, in practise there will always be a need to modify feature diagram, to extend feature library and, consequently, domain-specific language metamodel, modify generator and even to make some changes in generated code, there is no silver bullet. But we believe that our approach can provide better separation of concerns, provides better utilization of domain experts knowledge and expertise among a team. Summary of a process described above and relation between various tools and roles of developers is provided on Fig. 1.

This approach was implemented in a technology based on QReal DSM platform. QReal became an enabler technology because it provides easy and effective way to create visual editor for domain-specific languages that allows to create fully functional editor in less than an hour. It has

visual metaeditor, visual constraints definition tool, visual shape editor and a C++ library that allows to quickly specify generation rules. Feature diagram editor and generator that creates metamodel by feature diagrams were both implemented as plugins to QReal core. Note that feature diagram language is itself domain-specific language for the domain of domain analysis, so it was implemented using QReal metaeditor. The same metaeditor (including shape editor and constraints editor) is then used to tailor the generated metamodel of domain-specific language. Then metaeditor generator is used to generate yet another plugin to QReal that provides visual editor for created language. Then the generator is implemented by hand on C++ with Qt library<sup>2</sup> using generator creation library included in QReal. Then it is possible to create special distribution of QReal (using Qt Installer framework<sup>3</sup>) that includes only QReal core, editors for feature diagrams (at this point they are needed only as reference) and domain-specific language, generator and feature library, thus forming a complete technology that can be used to generate target applications.

#### IV. EVALUATION

For demonstration of the efficiency of proposed above approach there was implemented a model application for remote control of various robot models — “Joystick”. The main substantiation for implementing such application is that controlling different robot models requires different control elements. For example, one model can be controlled with only two pads, but another — with one pad and two buttons. Such application was implemented in C# for Windows Phone platform. Screenshots of this simple application are presented on Fig. 2.

As mentioned above, it was used QReal as DSM tool. A visual language was implemented there for describing feature models. Appropriate feature model for “Joystick” application family is proposed on Fig. 3. This feature model consists of explicit features, which are labeled green, and some feature groups, which are labeled blue. Type of arrow shows which feature is compulsory (shown as solid line with arrow on the end), which is compulsory but there is some alternative between them (shown as dash line with an arrow on the end), and optional features, which may be implemented but not compulsory (shown as dash line with a circle on the end).

<sup>2</sup>Qt library home page, URL: <http://www.qt.io/> (03.04.2016)

<sup>3</sup>Qt Installer Framework home page, URL: <https://wiki.qt.io/Qt-Installer-Framework> (03.04.2016)

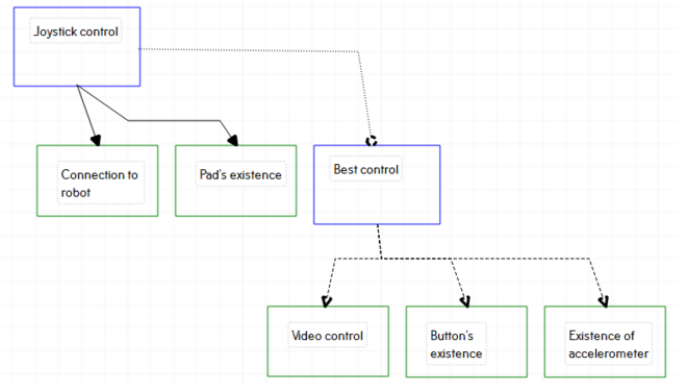


Figure 3: Feature model for “Joystick” application family.

Based on this feature model a metamodel for future visual language was generated, which is required for building different models for different configurations. Generated metamodel is presented on Fig. 4. As it can be seen, metamodel is very simple. At this stage we can propose that entities, such as “buttons” and “pads”, may have a property “Quantity”. In addition, we can specify images for these entities, which will be shown in visual language.

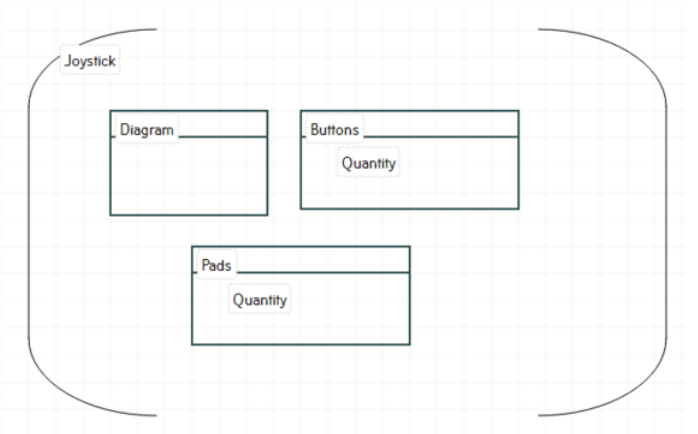


Figure 4: Metamodel of visual language for “Joystick” application family.

Then with the aid of QReal tool a visual language was generated. Example of generated visual language is demonstrated on Fig. 5. It can be seen that in visual language editor we can specify “Quantity” property, explicitly noting the concrete number of pads.



Figure 2: Screenshots of “Joystick” application.

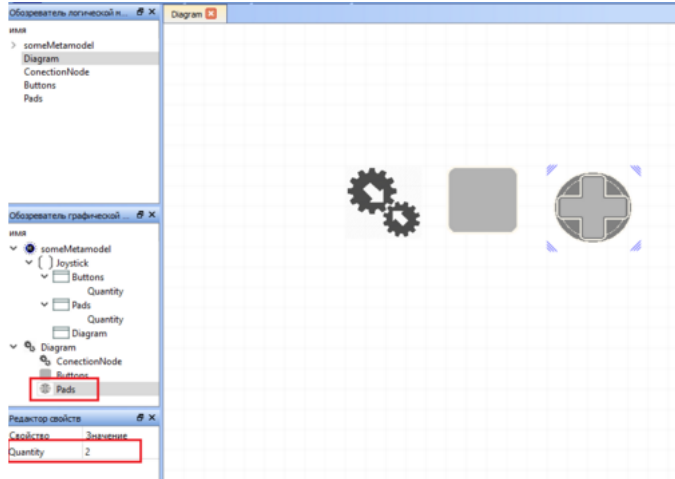


Figure 5: Visual language for “Joystick” application family.

As can be seen, example is quite simple for demonstrating extensive possibilities of the approach proposed above. At present there is no rigorous evaluation of the proposed process. Also, cohesive and consistent technology for creating application family based on domain analysis is not implemented yet, here we have described a concept-proof prototype. Therefore, this work requires more detailed explorations.

## V. CONCLUSION

The problem of not using domain analysis result for further generation of some entities for software development process was stated. There were considered some formal domain analysis approaches and we concluded that creation of feature diagrams is the most elegant decision for domain analysis that can be conducted by domain expert, i.e. non-programmer, maybe in collaboration with system analysts. Moreover, there was discussed one of the possible solutions, which is presented by Estublier, we specify some problems of such method. We suggested our own approach for creation of application family in one domain based on domain analysis. Thus, some target applications can be implemented even by non-programmers using domain-specific language with configuring features from library. Also, there was some evaluation of this approach, where

we pointed out that this example remains many questions because of its simplicity.

## REFERENCES

- [1] Tolvanen J.-p., Kelly S. Model-Driven Development Challenges and Solutions // *Modelsworld* 2016. 2016. P. 711–719.
- [2] Baker P., Loh S., Weil F. Model-driven engineering in a large industrial context — Motorola case study // *MoDELS’05: Proceedings of the 8th international conference on Model Driven Engineering Languages and Systems*. Berlin: Springer, 2005. P. 476–491.
- [3] A software engineering experiment in software component generation / R. Kieburtz, L. McKinney, J. Bell et al. // *Proceedings of the 18th international conference on Software engineering*. Washington, DC, USA: IEEE Computer Society, 1996. P. 542–552.
- [4] Kelly S., Tolvanen J.-P. Visual domain-specific modeling: Benefits and experiences of using metaCASE tools // *International Workshop on Model Engineering*, at ECOOP. 2000. URL: [http://dsmforum.org/papers/Visual\\_domain-specific\\_modelling.pdf](http://dsmforum.org/papers/Visual_domain-specific_modelling.pdf).
- [5] Tolvanen J.-P., Pohjonen R., Kelly S. Advanced tooling for domain-specific modeling: MetaEdit+ // *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM’07)*. 2007. URL: <http://www.dsmforum.org/events/DSM07/papers/tolvanen.pdf>.
- [6] Tolvanen J.-P. and Kelly S. MetaEdit+: defining and using integrated domain-specific modeling languages // *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications / ACM*. New York, NY, USA: ACM, 2009. P. 819–820.
- [7] Kelly S., Tolvanen J.-P. *Domain-specific modeling: enabling full code generation*. Hoboken, New Jersey, USA: Wiley-IEEE Computer Society Press, 2008. P. 444.
- [8] Gronback R. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Stoughton, Massachusetts, USA: Addison-Wesley, 2009. P. 736.
- [9] Viovic V., Maksimovic M., Perisic B. Sirius: A rapid development of DSM graphical editor // *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. Los Alamitos, CA, USA: IEEE Computer Society, 2014. P. 233–238.
- [10] *Domain-specific development with Visual Studio DSL Tools* / S. Cook, G. Jones, S. Kent et al. Crawfordsville, Indiana, USA: Addison-Wesley, 2007. P. 576.
- [11] Koznov D. *Process Model of DSM Solution Development and Evolution for Small and Medium-Sized Software Companies* // *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International / IEEE*. 2011. P. 85–92.
- [12] QReal DSM platform-An Environment for Creation of Specific Visual IDEs / A. Kuzenkova, A. Deripaska, T. Bryksin et al. // *ENASE 2013 — Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering*. Setubal, Portugal: SciTePress, 2013. P. 205–211.
- [13] Rugaber S. *Domain analysis and reverse engineering* // *White Paper*, January. 1994.

- [14] Prieto-Diaz R. Domain analysis for reusability // Software reuse: emerging technology / IEEE Computer Society Press. 1988. P. 347–353.
- [15] Ferré X., Vegas S. An evaluation of domain analysis methods // 4th CASE/IFIP8 International Workshop in Evaluation of Modeling in System Analysis and Design / Citeseer. 1999. P. 2–6.
- [16] Mernik M., Heering J., Sloane A. M. When and how to develop domain-specific languages // ACM computing surveys (CSUR). 2005. Vol. 37, no. 4. P. 316–344.
- [17] Arango G. Domain analysis methods // Software Reusability. 1994. P. 17–49.
- [18] DARE: Domain analysis and reuse environment / W. Frakes, R. Prieto, C. Fox et al. // Annals of Software Engineering. 1998. Vol. 5, no. 1. P. 125–141.
- [19] Taylor R. N., Tracz W., Coglianese L. Software development using domain-specific software architectures // ACM SIGSOFT Software Engineering Notes. 1995. Vol. 20, no. 5. P. 27–38.
- [20] Falbo R. d. A., Guizzardi G., Duarte K. C. An ontological approach to domain engineering // Proceedings of the 14th international conference on Software engineering and knowledge engineering / ACM. 2002. P. 351–358.
- [21] Feature-oriented domain analysis (FODA): Tech. Rep.: / K. C. Kang, S. G. Cohen, J. A. Hess et al.: DTIC Document, 1990.
- [22] Estublier J., Vega G. Reuse and variability in large software applications // ACM SIGSOFT Software Engineering Notes. 2005. Vol. 30, no. 5. P. 316–325.
- [23] An approach and framework for extensible process support system / J. Estublier, J. Villalobos, L. Anh-Tuyet et al. // Software Process Technology. Springer, 2003. P. 46–61.
- [24] The Variability Model of The Linux Kernel / S. She, R. Lotufo, T. Berger et al. // VaMoS. 2010. Vol. 10. P. 45–51.

# Language for Describing Templates for Test Program Generation for Microprocessors

Andrei Tatarnikov

Institute for System Programming of the Russian Academy of Sciences (ISP RAS)

25 Alexander Solzhenitsyn st., Moscow, 109004, Russian Federation

Email: andrewt@ispras.ru

**Abstract**—Test program generation and simulation is the most widely used approach to functional verification of microprocessors. High complexity of modern hardware designs creates a demand for automated tools that are able to generate test programs covering non-trivial situations in microprocessor functioning. The majority of such tools use test program templates that describe scenarios to be covered in an abstract way. This provides verification engineers with a flexible way to describe a wide range of verification tasks with minimum effort. Test program templates are developed in special domain-specific languages. These languages must fulfill the following requirements: (1) be simple enough to be used by verification engineers with no sufficient programming skills; (2) be applicable to various microprocessor architectures and (3) be easy to extend with facilities for describing new types of verification tasks. The present work discusses the test program template description language used in the reconfigurable and extensible test program generation framework MicroTESK being developed at ISP RAS. It is a flexible Ruby-based language that allows describing a wide range of test generation tasks in terms of hardware abstractions. The tool and the language have been applied in industrial projects dedicated to verification of MIPS and ARM microprocessors.

## I. INTRODUCTION

*Functional verification* is acknowledged to be the bottleneck in microprocessor design cycle. According to various estimates, it accounts for more than 70% of overall project time and resources. In the current industrial practice, function verification mainly relies on *test program generation (TPG)* which is done by special automation tools [1]. Generated *test programs (TP)* are instruction sequences aimed to trigger certain events in the microprocessor design under verification. TPG tools are aimed to provide a high level of test coverage by applying a rich set of generation methods. As modern microprocessors are getting more and more complex, new more advanced methods emerge. A common problem for TPG tool developers is how to overcome the complexity and make it easy to apply the growing set of methods to a wide range of microprocessor designs.

One of possible ways to increase the flexibility of a TPG tool is to separate generation logic from descriptions of test cases. This method is known as *template-based* generation. The key idea of the method is that test programs are generated on the basis of abstract descriptions called *test program templates* or *test templates (TTs)*. The method helps generate high-quality tests directed towards specific situations or classes of situations. TTs specify methods to be used for constructing

instruction sequences and constraints on instruction operand values which must be satisfied to make certain events to fire. Test data are generated by finding random solutions to the given constraint systems. Such approach is usually referred to as *constraint-based* random generation [2]).

The *template-based* approach is implemented in a number of TPG tools including MicroTESK [3], a reconfigurable [4] and extensible [5] TPG framework being developed at ISP RAS. The framework uses *formal specifications* to construct TPG tools for specific microprocessor designs. A constructed TPG tool is separated into two main components: (1) an architecture-independent test generation core and (2) an architecture specification, or a model. The approach called *model-based* [1] helps significantly reduce the efforts to support a new microprocessor architecture by reusing the core. The core is designed as a set of generation engines which can be easily extended with plugins implementing new TPG methods. Test programs are generated by processing TTs that describe verification tasks in terms of the model and the generation methods implemented by the core.

This paper describes the *test template description language (TTDL)* used in MicroTESK. This is a domain-specific language implemented as a set of Ruby [6] libraries, which is easy adaptable to changing configurations. Facilities for describing instruction calls for a specific ISA are dynamically added and are based on information provided by the model. Also, the MicroTESK TTDL provides a rich set of facilities for describing verification tasks which are common for all microprocessor configurations. When MicroTESK is extended with new TPG methods, support for these features is added in the TTDL by providing new Ruby libraries.

The rest of the paper is divided into five sections. Section II contains a brief survey of the existing TPG tools that follow the template-based approach. Section III formulates the requirements for a TTDL imposed by MicroTESK that led to creating the described TTDL. Section IV provides a detailed description of the architecture and facilities of the MicroTESK TTDL. Section V contains a case study of applying the TTDL for describing test cases in industrial projects. Section VI discusses the results and outlines directions of future research and development.

## II. RELATED WORK

Functional verification has always been a major issue for the research community. Over the last decades, a lot of TPG methods and tools have emerged. The template-based approach described in this paper has been applied in a number of tools developed by different teams. This section gives an overview of the most significant of existing TPG tools and discusses strong and weak points of their TTDLs.

IBM Research has been one of the major contributors in the field of TPG for microprocessors during the last decades. Genesys-Pro [1], one of their most recent tools, uses TTs to describe TPG tasks as constraint satisfaction problems (CSP) [2] and generates test data by solving these CSPs. Constraints can be used to specify such aspects of functionality as boundary conditions, exceptions, cache hits/misses, etc. The TTDL used by Genesys-Pro is a completely independent domain-specific language which provides a rich set of features. The language features it offers can be divided into four groups: (1) basic instruction statements, (2) sequencing-control statements, (3) standard programming constructs, and (4) constraint statements. By combining these constructs, users can compose complex TTs with a degree of randomness varied from completely random to completely directed. The main advantage of the language is that it is designed for describing test scenarios and it does not confuse verification engineers with any unnecessary programming constructs. At the same time, being not based on existing languages, it does not take advantage of well-tried constructs that can help organize TTs into reusable libraries. This can be important as industrial testbenches usually contain thousands lines of code. Also, it is unclear how easy the language can be extended with new constructs for describing new types of TPG tasks.

Another company that has made a significant contribution in development of TPG tools is Obsidian Software (now acquired by ARM) [7]. Their tool RAVEN (Random Architecture Verification Engine) [8] generates random and directed tests based on TTs. Test templates are focused on coverage grids and use constraints to formulate specific coverage goals. There is no detailed information available on this technology. It is known that TTs can be either generated by the tool's GUI or created as text. The language must suit well for the TPG tasks that can be accomplished with RAVEN. However, the question whether it is suitable for more general tasks stays open.

Also, Samsung Electronics created a TPG framework called RDG (Random Diagnostics Generator) [9] for testing reconfigurable processors. It uses TTs created in the C++ language to specify instructions that will be used in a TP and constraints on their input values that should be satisfied in order to meet testing goals. This approach takes advantage of power and performance of C++, but requires solid programming skills which are not common for verification engineers.

Finally, MicroTESK [3] version 1.0 used TTs written using Java libraries [10]. This is not convenient as verification engineers are forced to deal with Java abstractions such as classes and interfaces, which are not related to verification

tasks. Moreover, details of language implementation must be hidden from users in order to be able to change it without breaking existing TTs. This motivated to create a new domain-specific language for the new version of MicroTESK.

## III. REQUIREMENTS FOR TTDL

Requirements for a TTDL can be divided in two groups: (1) general requirements for a TTDL; (2) requirements related to integration into the MicroTESK framework. Let us first consider the general requirements that are common for all TTDLs. A TTDL used to describe scenarios for random and directed tests must provide facilities:

- 1) to describe instructions calls and data definitions using syntax similar to the one used in assembly code;
- 2) to manage memory allocations in the same way as in the assembly language;
- 3) to fill memory with data generated according to specific rules;
- 4) to compose instruction sequences using a wide range of methods (random, combinatorial, etc.) and to merge these sequences;
- 5) to specify random values and the degree of their randomness described by distributions;
- 6) to select instructions at random with the specified degree of randomness;
- 7) to specify constraints on instruction arguments;
- 8) to describe initialization code that places generated test data to proper registers or memory addresses;
- 9) to specify code of self-checks that check validity of the resulting state of the microprocessor;
- 10) to describe exception handlers;
- 11) to specify conditions for generating different code depending on the context;
- 12) to insert comments and custom text into generated TPs;
- 13) to reuse existing TTs and their parts;
- 14) to split generated TPs into multiple files.

This list is not complete, but it is enough to conclude that the TTDL must be a domain-specific language that provides constructs for the listed facilities.

Another important consideration is that it must be integrated into MicroTESK. First of all, MicroTESK is written in Java and its generation engines operate with Java objects. Therefore, the result of TT processing must be a hierarchy of Java objects that then will be passed to TPG engines. The front-end of a TTDL processor can be implemented using two approaches: (1) creating a Java-based parser for the new language or (2) reusing an existing Java-based parser for one of the popular programming languages. A crucial requirement for the second approach is that the language must be easy to extend with new domain-specific constructs.

Now let us consider the requirements imposed by reconfigurability and extensibility of MicroTESK:

- 1) *Reconfigurability* means that it can be applied to microprocessors with different ISAs. Consequently, facilities used to describe instruction calls must be changeable.



Ideally, they must be added dynamically depending on the information provided in the model that describes the configuration of the design under verification.

- 2) *Extensibility* means that the set of supported TPG methods can be extended by adding plugins implementing new methods. Often it will require adding new constructs in the TTDL. Thus, it must be possible to dynamically add language constructs depending on the installed plugins.

In other words, a crucial requirement for the MicroTESK TTDL is the ability to dynamically change the set of supported language constructs. Obviously, changes in the tool configuration must not involve modification of the TTDL processor. Creating a flexible language processor from scratch is a challenging task. A simpler solution would be to reuse a parser of an existing language.

Having considered several possible alternatives, it was decided to use JRuby [11], a Java-based implementation of the Ruby language, as a front-end of the TTDL processor. Ruby was selected because of its support for *metaprogramming* [12], which allows adding new language features at runtime. Thus, the created TTDL combines basic programming constructs provided by the Ruby core with constructs for describing TTs provided by MicroTESK. The TTDL front-end is implemented as a set of Ruby libraries that define language facilities for the above mentioned requirements. Facilities that depend on the current configuration are dynamically added using metaprogramming.

It is also worth mentioning that scripting languages like Ruby are quite popular among verification engineers, who often use them to create inhouse test generators. So, another advantage of using Ruby is that it can make the TTDL easier to learn.

#### IV. TTDL DESCRIPTION

##### A. Language Processor Architecture

The job of the TTDL processor is to build a hierarchy of Java objects describing a TT and to pass it to the MicroTESK generation engines for further processing. The TTDL processor is divided into a *Ruby-based front-end* and *Java-based back-end*. The back-end is implemented as set of factories for creating Java objects that correspond to specific entities of a TT. The front-end is represented by Ruby libraries that provide language constructs for describing these entities and perform interaction with the back-end to build corresponding Java objects. In other words, a language feature is defined by a Ruby module that specifies its syntax and a Java module that describes corresponding entities and provides means of constructing them. New language features can be supported by providing corresponding modules.

The TTDL contains features that are configuration dependent. This includes facilities for describing instruction calls, which are determined by the model built by MicroTESK from ISA specifications. These language features are managed by a special Ruby module that uses metaprogramming to define

corresponding constructs at runtime based on the information provided by the model.

##### B. Test Template Structure

A TT is a program in Ruby which is executed by MicroTESK with the help of JRuby to build Java objects that formulate tasks for the TPG engines implemented by the tool core. More technically, it is a subclass of the `Template` base class provided by the MicroTESK library. All domain-specific language constructs are implemented as methods of this class. The `Template` class is not monolithic, it unites a set of Ruby modules responsible for various features into a single class. Language extensions are also implemented as modules to be included in the base class. Configuration-specific methods are dynamically defined when the class is loaded.

The listing below shows the structure of a TT class:

```
require ENV['TEMPLATE']

class MyTemplate < Template
  def initialize
    super
    # Initialize settings here
  end

  def pre
    # Place your initialization code here
  end

  def post
    # Place your finalization code here
  end

  def run
    # Place your testing task description here
  end
end
```

The first line imports the `Template` base class from the location specified by the `TEMPLATE` environment variable. The exact location depends on the configuration and is determined automatically.

Classes describing TTs define four methods:

- `initialize` - configures TT settings if there is a need to override the default;
- `pre` - defines ISA-specific constructs and specifies initialization code to be inserted in the beginning of TPs;
- `post` - specifies finalization code to be inserted in the end of TPs;
- `run` - contains descriptions of test cases to be generated.

The methods will be filled with constructs described further.

##### C. Managing Memory Allocation

It may be required to place code and data sections of generated TPs at specific memory locations. The assembly language provides special directives to accomplish this task. The TTDL offers similar constructs. An important note is that MicroTESK simulates TPs in the process of their generation. Consequently, these constructs not only specify directives to be placed into TPs, but also manage memory allocation in the simulator.

The TTDL provides the following methods for managing addresses, which are applicable to both code and data sections:

- `align` - aligns the allocation address by the amount `n` passed as an argument, which by default means  $2^n$  bytes.
- `org` - sets the allocation origin, which is required to increase the allocation address. It is possible to set an *absolute* or *relative* origin. The former can be specified as `org n` and means an offset by `n` bytes from the base virtual address. The latter can be specified as `org :delta=>n` and means an offset by `n` bytes from the most recent allocation address.
- `label` - associates the specified label with the current address.

The listed methods rely on the following TT settings:

- `align_format` - specifies textual format for the `align` directive;
- `org_format` - specifies textual format for the `org` directive;
- `base_virtual_address` - specifies the base virtual address for memory allocation;
- `base_physical_address` - specifies the base physical address for memory allocation;
- `alignment_in_bytes` - specifies how the alignment amount should be interpreted.

The first four settings are initialized with default values in the `initialize` method of the `Template` base class as shown below and can be changed in the current TT class:

```
@org_format = ".org 0x%x"
@align_format = ".align %d"
@base_virtual_address = 0x0
@base_physical_address = 0x0
```

The last setting is implemented as method that can be overridden to change its behavior:

```
def alignment_in_bytes(n)
  2 ** n
end
```

#### D. Defining Random Distributions

Many TPG tasks involve selection based on *random distribution*. The TTDL provides the following methods to define random distributions:

- `range` - creates an object describing a range of values and its weight, which are specified by the `value` and `bias` attributes. Values can be one of the following types:
  - *single* value;
  - *range* of values;
  - *array* of values;
  - *distribution* of values.

The `bias` attribute can be skipped which means default weight. Default weights are used to specify an even distribution based on ranges with equal weights.

- `dist` - creates an object describing a random distribution from a collection of ranges.

The code below illustrates how to create weighted distributions for integer numbers:

```
simple_dist = dist(
  range(:value => 0,      :bias => 25), # Value
  range(:value => 1..2,    :bias => 25), # Range
  range(:value => [3, 5, 7], :bias => 50) # Array
)

composite_dist = dist(
  range(:value=> simple_dist, :bias => 80), # Distribution
  range(:value=> [4, 6, 8],    :bias => 20) # Array
)
```

#### E. Describing Data Definitions

*Data definitions* are based on assembler-specific directives, which are not described by the microprocessor model and, therefore, must be configured in TTs. The configuration information includes textual format of the directives and mappings between data types used by the assembler and the microprocessor model. Data directives are configured using the `data_config` construct, which must be placed in the `pre` method. Here is an example:

```
data_config(:text=>".data", :target=>"MEM") {
  define_type :id=>:byte, :text=>".byte", :type=>card(8)
  define_type :id=>:half, :text=>".half", :type=>card(16)
  define_type :id=>:word, :text=>".word", :type=>card(32)

  define_space :id=>:space, :text=>".space", :fillWith=>0
  define_ascii :id=>:ascii, :text=>".ascii", :zero=>false
  define_asciiz :id=>:asciiz, :text=>".asciiz", :zero=>true
}
```

The `data_config` method has the following parameters:

- `text` - specifies the textual format of a directive that marks the beginning of a data section;
- `target` - specifies the memory array defined in the model to which data will be placed during simulation;
- `base_virtual_address` (optional, 0 by default) - specifies the base virtual address for data sections.

Distinct data directives are configured using special methods that must be called inside the `data_config` block. All of these methods share two common parameters: `id` and `text`. The first specifies the keyword to be used in a TT to address the directive and the second specifies how it will be printed into the TP. Here is the list of methods:

- `define_type` - defines a directive to allocate memory for a data element of the data type specified by the `type` parameter;
- `define_space` - defines a directive to allocate memory filled with a default value specified by the `fillWith` parameter;
- `define_ascii_string` - defines a directive to allocate memory for an ASCII string terminated or not terminated with zero depending on the `zero` parameter.

The above example defines directives `byte`, `half`, `word`, `ascii` (non-zero terminated string) and `asciiz` (zero terminated string) that place data in the memory array `MEM` defined in the microprocessor model.

Once data directives have been configured, data sections can be defined using the `data` construct. Data definitions can be of two kinds depending on the context:

- 1) *Global data* that are available to all test cases generated from the given TT. They are defined in the root of the pre or run methods. Global data are placed into the simulator's memory during initial processing of a TT.
- 2) *Test case level data* that are defined and used by specific test cases. Such data are placed into the simulator's memory when the test case is being generated.

The data method has two optional parameters:

- `global` - a flag that states that the data definition should be treated as global regardless of the context.
- `separate_file` - a flag that specifies whether the generated data definitions should be placed into a separate source code file.

Here is an example of a data definition:

```
data(:global => true, :separate_file => false) {
  org 0x00001000

  label :byte_values
  byte 1, 2, 3, 4

  label :word_values
  word 0xDEADBEEF, 0xBAADF00D
}
```

The above code defines global data: four byte values and two word values. Memory is allocated at offset 0x00001000. Data values are aligned by their size (1 and 4 bytes). Labels `byte_values` and `word_values` point at the beginning of the byte and the word arrays correspondingly.

### F. Describing Instruction Calls

To describe instruction calls, the TTDL provides runtime methods that are defined using the metaprogramming facilities of Ruby on the basis of information provided by the model. Methods have the same names and parameters as operations describing corresponding instructions, which are defined in ISA specifications. Operations use parameters of three kinds:

- 1) *Immediate values* that represent constants.
- 2) *Addressing modes* that encapsulate logic of reading and writing data to memory resources. Usually they provide access to registers or memory.
- 3) *Operations* that specify operations to be performed as a part of execution of the current operation. They are used to describe complex instructions composed of several operations (e.g. VLIW instructions).

For example, a call to the `add` instruction from the MIPS ISA [13], which adds two general-purpose registers `t0` (\$8), `t1` (\$9) and `t2` (\$10) described by the `reg` addressing mode, can be specified in the following way:

```
add reg(8), reg(9), reg(10)
```

The TTDL supports creating *aliases* for addressing modes and operations invoked with certain arguments. Aliases help make TTs more human-readable. They are created by defining Ruby functions with corresponding names. The code below shows how to create aliases for the registers from the previous example:

```
def t0 reg(8) end
def t1 reg(9) end
def t2 reg(10) end
```

Now the arguments of the `add` instruction can be specified using aliases:

```
add t0, t1, t2
```

Also, the TTDL provides the `pseudo` function that can be used to specify calls to *pseudo instructions* that do not have corresponding operations in ISA specifications. They print user-specified text, but are not simulated by the generator. Here is an example:

```
pseudo 'syscall'
```

### G. Defining Groups

Addressing modes and operations can be organized into *groups*. Groups are used when it is required to randomly select an addressing mode or operation from the specified set. Groups can be defined in ISA specifications or in TTs. To do this in TTs, the `define_mode_group` and `define_op_group` functions are used. Both functions take the name and distribution arguments that specify the group name and the distribution used to select its items.

For example, the code below defines an instruction group called `alu` that contains instructions `add`, `sub`, `and`, `or`, `nor`, and `xor` selected randomly according to the specified distribution:

```
alu_dist = dist(
  range(:value => 'add', :bias => 40),
  range(:value => 'sub', :bias => 30),
  range(:value => ['and', 'or', 'nor', 'xor'], :bias => 30)
)

define_op_group('alu', alu_dist)
```

The following code specifies three calls that use instructions randomly selected from the `alu` group:

```
alu t0, t1, t2
alu t3, t4, t5
alu t6, t7, t8
```

### H. Describing Instruction Call Sequences

Instruction call sequences are described using block-like structures. Each block specifies a sequence or a collection of sequences. Blocks can be nested to construct complex sequences. The algorithm used for sequence construction depends on the type and the attributes of a block.

An individual instruction call is considered a primitive block describing a single sequence that consists of a single instruction call. A single sequence that consists of multiple calls can be described using the `sequence` or the `atomic` construct. The difference between the two is that an atomic sequence is never mixed with other instruction calls when sequences are merged. The code below demonstrates how to specify a sequence of three instruction calls:

```
sequence {
  add t0, t1, t2
  sub t3, t4, t5
  or t6, t7, t8
}
```

A collection of sequences that are processed one by one can be specified using the `iterate` construct. For example, the code below describes three sequences consisting of one instruction call:

```
iterate {
  add t0, t1, t2
  sub t3, t4, t5
  or t6, t7, t8
}
```

Sequences can be combined using the `block` construct. The resulting sequences are constructed by sequentially applying the following engines to sequences returned by nested blocks:

- `combinator` - builds combinations of sequences returned by nested blocks. Each combination is a tuple of length equal to the number of nested blocks.
- `permutator` - modifies combinations returned by `combinator` by rearranging some sequences.
- `compositor` - merges (multiplexes) sequences in a combination into a single sequence preserving the initial order of instructions calls in each sequence.
- `rearranger` - rearranges sequences constructed by `compositor`.
- `obfuscator` - modifies sequences returned by `rearranger` by permuting some instruction calls.

Each engine has several implementations based on different methods. It is possible to extend the list of supported methods with new implementations. Specific methods are selected by specifying corresponding block attributes. When they are not specified, default methods are applied. The format of a block structure for combining sequences looks as follows:

```
block (
  :combinator => 'combinator-name',
  :permutator => 'permutator-name',
  :compositor => 'compositor-name',
  :rearranger => 'rearranger-name',
  :obfuscator => 'obfuscator-name') {

  # Block A. 3 sequences of length 1: {A11}, {A21}, {A31}
  iterate { A11; A21; A31 }

  # Block B. 2 sequences of length 2: {B11, B12}, {B21, B22}
  iterate { sequence { B11, B12 }; sequence { B21, B22 } }

  # Block C. 1 sequence of length 3: {C11, C12, C13}
  iterate { sequence { C11; C12; C13 } }
}
```

The default method names are: `diagonal` for `combinator`, `catenation` for `compositor`, and `trivial` for `permutator`, `rearranger` and `obfuscator`. Such a combination of engines describes a collection of sequences constructed as a concatenation of sequences returned by nested blocks. For example, sequences constructed for the block in the above example will be as follows: {A11, B11, B12, C11, C12, C13}, {A21, B21, B22, C11, C12, C13} and {A31, B11, B12, C11, C12, C13}.

## I. Specifying Test Situations

*Test situations* are associated with specific instruction calls and specify methods used to generate their input data. There is a wide range of data generation methods implemented by various data generation engines. Test situations are specified using the `situation` construct. It takes the situation name and a map of optional attributes that specify situation-specific parameters. For example, the following line of code causes input registers of the `add` instruction to be filled with zeros:

```
add t1, t2, t3 do situation('zero') end
```

When no situation is specified, a default situation is used. This situation places random values into input registers. It is possible to assign a custom default situation for individual instructions and instruction groups with the `set_default_situation` function. For example:

```
set_default_situation 'add' do situation('zero') end
```

Situations can be selected at random. The selection is based on a distribution. This can be done by using the `random_situation` construct. For example:

```
sit_dist = dist(
  range(:value => situation('add.overflow')),
  range(:value => situation('add.normal')),
  range(:value => situation('zero')),
  range(:value => situation('random', :dist => int_dist))
)

add t1, t2, t3 do random_situation(sit_dist) end
```

Unknown immediate arguments that should have their values generated are specified using the `"_"` symbol. For example, the code below states that a random value should be added to a value stored in a random register and the result should be placed to another random register:

```
addi reg(_), reg(_), _ do situation('random') end
```

## J. Selecting Registers

Unknown immediate arguments of addressing modes are a special case and their values are generated in a slightly different way. Typically, they specify register indexes and are bounded by the length of register arrays. Often such indexes must be selected from a specific range taking into account previous selections. For example, registers are allocated at random and they must not overlap. To be able to solve such tasks, all values passed to addressing modes are tracked. The allowed value range and the method of value selection are specified in configuration files. Values are selected using the specified method before the instruction call is processed by the engine that generates data for the test situation. The selection method can be customized by using the `mode_allocator` function. It takes the allocation method name and a map of method-specific parameters. For example, the following code states that the output register of the `add` instruction must be a random register which is not used in the current test case:

```
add reg(_ mode_allocator('free')), t0, t1
```

Also, the TTDL allows customizing the allowed range for selected values. It is possible to exclude some elements from the range by using the `exclude` attribute or to provide a new range by using the `include` attribute. For example:

```
add reg(_ :exclude=>[1, 5, 7]), t0, t1
add reg(_ :include=>8..15), t0, t1
```

Addressing modes with specific argument values can be marked as free using the `free_allocated_mode` function. To free all allocated addressing modes, the `free_all_allocated_modes` function can be used.

### K. Describing Preparators

*Preparators* describe instruction sequences that place data into registers or memory accessed via the specified addressing mode. They are inserted into TPs to set up the initial state of the microprocessor required by test situations. It is possible to overload preparators for specific cases (value masks, register numbers, etc). Preparators are defined in the `pre` method using the `preparator` construct, which uses the following parameters describing conditions under which it is applied:

- `target` - the name of the target addressing mode;
- `mask` (optional) - the mask that should be matched by the value in order for the preparator to be selected;
- `arguments` (optional) - values of the target addressing mode arguments that should be matched in order for the preparator to be selected;
- `name` (optional) - the name that identifies the current preparator to resolve ambiguity when there are several different preparators that have the same target, mask and arguments.

It is possible to define several variants of a preparator which are selected at random according to the specified distribution. They are described using the `variant` construct. It has two optional parameters:

- `name` (optional) - identifies the variant to make it possible to explicitly select a specific variant;
- `bias` - specifies the weight of the variant, can be skipped to set up an even distribution.

Here is an example of a preparator what places a value into a 32-bit register described by the `REG` addressing mode and two its special cases for values equal to `0x00000000` and `0xFFFFFFFF`:

```
preparator(:target => 'REG') {
  variant(:bias => 25) {
    data {
      label :preparator_data
      word value
    }

    la at, :preparator_data
    lw target, 0, at
  }

  variant(:bias => 75) {
    lui target, value(16, 31)
    ori target, target, value(0, 15)
  }
}

preparator(:target => 'REG', :mask => '00000000') {
```

```
xor target, zero, zero
}

preparator(:target => 'REG', :mask => 'FFFFFFF') {
  nor target, zero, zero
}
```

Code inside the `preparator` block uses the `target` and `value` functions to access the target addressing mode and the value passed to the preparator.

The TTDL provides the `prepare` function to explicitly insert preparators into TPs. It can be used to create composite preparators. The function has the following arguments:

- `target` - specifies the target addressing mode;
- `value` - specifies the value to be written;
- `attrs` (optional) - specifies the preparator name and the variant name to select a specific preparator.

For example, the following line of code places value `0xDEADBEEF` into the `t0` register:

```
prepare t0, 0xDEADBEEF
```

### L. Describing Self-Checks

TPs can include self-checks that check validity of the microprocessor state after a test case has been executed. These checks are instruction sequences inserted in the end of test cases which compare values stored in registers with expected values. If the values do not match control is transferred to a handler that reports an error. Expected values are produced by the MicroTESK simulator. Self-check are described using the `comparator` construct which has the same features as the `preparator` construct, but serves a different purpose. Here is an example of a comparator for 32-bit registers and its special case for value equal to `0x00000000`:

```
comparator(:target => 'REG') {
  prepare target, value
  bne at, target, :check_failed
  nop
}

comparator(:target => 'REG', :mask => "00000000") {
  bne zero, target, :check_failed
  nop
}
```

### M. Describing Test Cases

A TP can be described by the following formula:  

$$\Pi = \Pi_{start} \cdot \{\langle \pi_{start}, x_i, \pi_{stop} \rangle\}_{i=1,n} \cdot \Pi_{stop}$$
 [10],  
 where:

- $\Pi_{start}$  is a TP prologue that consists of instructions aimed for microprocessor initialization;
- $\langle \pi_{start}, x_i, \pi_{stop} \rangle$  is a test case that specifies an individual stimulus and consists of:
  - $\pi_{start}$  is a test case prologue that performs all necessary preparations for the test case;
  - $x_i$  is a test case action that contains the main code of the test case;
  - $\pi_{stop}$  is a test case epilogue that performs finalization actions for the test case such as self-checks.

- $\Pi_{stop}$  is a TP epilogue that consists of instructions aimed for microprocessor finalization;
- $n$  is the number of test cases in a TP.

The TTDL provides means of describing each part of a TP.  $\Pi_{start}$  and  $\Pi_{stop}$  are described in the `pre` and `post` methods of a TT class correspondingly. Test cases are described in the `run` method.

Test cases are described by block constructs specifying one or more sequences of instruction calls. Each sequence is a separate test case. It is possible to process a block multiple times. This makes sense when sequences use randomization. In this case, it results in different test cases based on the same description. For example, the code below describes five test cases based on the same sequence of three calls. Input data for the calls are generated at random and will be different for all test cases.

```
def run
  sequence {
    add t0, t1, t2
    sub t3, t4, t5
    or t6, t7, t8
  }.run 5
end
```

$\pi_{start}$  that contains preparators for input registers and  $\pi_{stop}$  that contains self-checks will be generated by the tool automatically. Also, it is possible to specify additional prologue and epilogue for test cases. They will be inserted between automatically generated prologue and epilogue and main code of the test cases. They are specified using the `prologue` and `epilogue` blocks nested into the `sequence` block. The syntax looks like this:

```
sequence {
  prologue { ... }
  ...
  epilogue { ... }
}.run n
```

When instruction sequences are merged by nesting blocks, prologue and epilogue of nested blocks wrap sequences returned by these blocks.

Test cases can be processed by different TPG engines. A specific engine can be selected by passing the `engine` parameter to the block construct that describes the test cases.

## N. Describing Exception Handlers

TGs must contain handlers of exceptions that may occur during their execution. Exception handlers are described using the `exception_handler` construct. This description is also used by the MicroTESK simulator to handle exceptions. Separate exception handlers are described using the `section` construct nested into the `exception_handler` block. The `section` function has two arguments: `org` that specifies the handler's location in memory and `exception` that specifies names of associated exceptions. For example, the code below describes a handler for the `IntegerOverflow`, `SystemCall` and `Breakpoint` exceptions which resumes execution from the next instruction:

```
exception_handler {
  section(:org => 0x380, :exception => ['IntegerOverflow',
                                      'SystemCall',
                                      'Breakpoint']) {

    mfc0 ra, cop0(14)
    addi ra, ra, 4
    jr ra
    nop
  }
}
```

## O. Printing Text

TGs are printed in textual form to source code files. The printed text includes various supplementary messages such as comments and separators. They are generated by MicroTESK engines or specified by users in TGs. The format of printed text is set up using the following settings:

- `sl_comment_starts_with` - starting characters for single-line comments. Default value is `"/"`.
- `ml_comment_starts_with` - starting characters for multi-line comments. Default value is `"/*"`.
- `ml_comment_ends_with` - terminating characters for multi-line comments. Default value is `"*/"`.
- `indent_token` - indentation token. Default value is `"\t"`.
- `separator_token` - token used in separator lines. Default value is `"="`.

The settings are initialized with default values in the `initialize` method of the `Template` class can be re-defined in the `initialize` method of a TG.

The TTDL provides functions for printing custom text messages. Text messages are printed either into the generated source code or into the simulator log. Here is the list of supported functions:

- `newline` - adds the new line character into the TG;
- `text` - adds text into the TG;
- `trace` - prints text into the simulator execution log;
- `comment` - adds a comment into the TG;
- `start_comment` - starts a multi-line comment;
- `end_comment` - ends a multi-line comment.

The `text`, `trace` and `comment` functions print formatted text. They take a format string and an array of objects to be printed, which can be constants or memory locations. To specify locations to be printed (registers, memory), the `location` function should be used. It takes the name of the memory array and the index of the selected element. For example, the code below prints a constant value and a value stored in a register in the hexadecimal format:

```
text 'Constant: 0x%X', 0xDEADBEEF
text 'Register: 0x%X', location('GPR', 8)
```

## V. CASE STUDY

MicroTESK and its TTDL have been applied in industrial projects to generate TGs for MIPS64 [13] and ARMv8 [14] microprocessors. Table I provides characteristics of the MIPS64 and ARMv8 specifications used to configure MicroTESK for generating TGs for these designs.

TABLE I  
INDUSTRIAL APPLICATION OF THE PROPOSED TTDL AND SUPPORTING TOOL.

Project	MIPS64	ARMv8
Number of instructions	218	394
ISA specification size (lines of code)	4300	9000
MMU specification size (lines of code)	500	2000
Efforts (person-months)	4	9

Created tests include:

- tests for arithmetical instructions;
- tests for floating-point instructions;
- tests for branch instructions;
- tests for memory access instructions.

To describe tests for branch and memory instruction, the TTDL was extended with additional constructs based on existing ones. The language was evolving in the process of working on the projects. Some features were changed and some were added. A number of language features came as requirements from customers. The approach based on using dynamic languages such as Ruby to create TTDLs has proved its flexibility. The TTDL allowed describing test cases in a format which is maximally close to assembly language for corresponding microprocessors. This allows verification engineers to concentrate on verification problems instead of issues related to the use of a specific programming language.

## VI. CONCLUSION

A concept of a TTDL for a reconfigurable and extensible TPG framework has been considered. The proposed solution was implemented in the MicroTESK [3] framework. The developed TTDL is based on the Ruby [6] language and uses its metaprogramming facilities to dynamically add configuration-dependent language constructs. The language is integrated into MicroTESK, which is a Java-based tool, with the help of JRuby [11]. Facilities of the TTDL can be extended by adding new Ruby libraries.

Directions for further research and development are to apply the described principles to create TTDLs based on other programming languages. First of all, it is Python and its Java-based implementation called Jython. It provides facilities similar to those of Ruby and is also popular among verification engineers. For this reason, it would be advantageous to provide a Python-based version of the TTDL for those who are more comfortable with this language.

Another task is development of a TTDL based on C++. It will be a part of a large research project dedicated to *on-line* generation. An on-line TPG tool is represented by a binary image with basic functions of an operating system, which is loaded directly to a microprocessor chip where it generates and executes test stimuli. The tool will be created by MicroTESK

from C++ libraries based on formal specifications. For further unification of TPG tools, it is important that TTs for on-line generation are developed using the same principles.

## REFERENCES

- [1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimón, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. Design & Test of Computers, 2004. pp. 84–93.
- [2] Y. Naveh, M. Rimón, I. Jaeger, Y. Katz, M. Vinov, E. Marcus and G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification*. AI Magazine, Volume 28, Number 3, 2007, pp. 13–30.
- [3] *MicroTESK page* – <http://forge.ispras.ru/projects/microtesk>
- [4] A. Kamkin, E. Kornychin, D. Vorobyev. *Reconfigurable Model-Based Test Program Generator for Microprocessors*. International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47–54.
- [5] A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. *Extensible Environment for Test Program Generation for Microprocessors*. Programming and Computer Software, 40(1), 2014. pp. 1–9.
- [6] *Ruby site* – <http://www.ruby-lang.org>
- [7] E.A. Poe. *Introduction to Random Test Generation for Processor Verification*. Obsidian Software, 7 pp, 2002.
- [8] *RAVEN test program generator* – <http://www.slideshare.net/DVClub/introducing-obsidian-software-and-ravengcs-for-powerpc>
- [9] Seonghun Jeong, Youngchul Cho, Daeyong Shin, Changyeon Jo, Yenjo Han, Soojung Ryu, Jeongwook Kim, and Bernhard Egger. *Random Test Program Generation for Reconfigurable Architectures*. 13th International Workshop on Microprocessor Test and Verification (MTV), 2012, 6 p.
- [10] A. Kamkin. *Generatsiya testovykh programm dlya mikroprotssessorov [Test Program Generation for Microprocessors]*. Trudy ISP RAN [Proceedings of ISP RAS], Volume 14, Part 2, 2008, pp. 2363 (in Russian).
- [11] *JRuby site* – <http://www.jruby.org>
- [12] Flanagan D., Matsumoto Y. *The Ruby Programming Language*. O'Reilly Media, Sebastopol, 2008.
- [13] *MIPS64<sup>TM</sup> Architecture For Programmers*. Volume II: The MIPS64<sup>TM</sup> Instruction Set, Document Number: MD00087, Revision 2.00, June 9, 2003.
- [14] *ARM Architecture Reference Manual*. ARM DDI 0487A.f, ARM Corporation, 2015. 5886 p.



# Specification-Based Test Program Generation for MIPS64 Memory Management Units

Alexander Kamkin, Artem Kotsynyak

Institute for System Programming of the Russian Academy of Sciences (ISP RAS)

25 Alexander Solzhenitsyn st., Moscow, 109004, Russian Federation

Email: {kamkin, kotsynyak}@ispras.ru

**Abstract**—In this paper, a tool for automatically generating test programs for MIPS64 memory management units is described. The solution is based on the MicroTESK framework being developed at ISP RAS. The tool consists of two parts: an architecture-independent test program generation core and MIPS64 MMU specifications. Such separation is not a new principle in the area – it is applied in a number of industrial test program generators, including IBM’s Genesys-Pro. The main distinction is in how specifications are represented, what sort of information is extracted from them, and how that information is exploited. In the suggested approach, specifications comprise descriptions of the memory access instructions, loads and stores, and definition of the memory management mechanisms such as translation lookaside buffers, page tables, and cache units. The tool analyzes the specifications and extracts the execution paths and inter-path dependencies. The extracted information is used to systematically enumerate test programs for a given user-defined template. Test data for a particular program are generated by using symbolic execution and constraint solving techniques.

## I. INTRODUCTION

A computer memory is known to be a complex hierarchy of data storage devices varying in volume, latency and price. In addition to registers and main memory, microprocessors include a multi-level cache memory and address translation buffers. The set of devices responsible for handling memory accesses is referred to as a *memory subsystem* or a *memory management unit* (MMU). Being one of the key microprocessor components, the memory subsystem is strongly required to be correct and reliable. Due to the complicated structure of the memory, the number of situations that can occur in processing load and store instructions is huge; this makes it improbable to verify the subsystem “manually”.

It is widely accepted that *test program generation* (TPG) is an essential approach to microprocessor verification [1]. The problem is how to overcome the complexity and at the same time provide acceptable test coverage. It is a fallacy that (naïve) random TPG is a good way to optimize testing [2]. A better solution, we think, is a *specification-based approach* [1]. A TPG tool consists of two components: (1) an architecture-independent test generation *core* and (2) an architecture *specification*, or *model*. The approach reduces the efforts to create a generator by reusing the core – the only thing one needs to develop is a specification.

There exist a number of tools implementing the paradigm mentioned above [1], [3], [4]. However, only few of them are distributed under open licenses. ISP RAS’s MicroTESK [5]

is one of those few. The tool uses a dialect of the nML language [6] for specifying *instruction set architectures* (ISA) and an extensible set of dedicated languages for specifying particular microarchitectural features, including, first of all, a memory management. In this work, we would like to share our experience in creating a MicroTESK-based TPG for verifying MIPS64 MMUs [8].

The remainder of this paper is divided into four sections. Section 2 contains a brief survey of the existing approaches to TPG for MMUs. Section 3 presents the MicroTESK framework and its facilities aimed at MMU specification and testing. Section 4 studies application of the TPG approach for MIPS64 MMU. Section 5 discusses the results of the work and outlines directions of future research and development.

## II. RELATED WORK

There are several TPG tools based on formal specifications of memory subsystems. IBM’s DeepTrans [9] uses a dedicated specification language. Address translation is depicted as a *directed acyclic graph* (DAG) whose vertices correspond to the process stages and whose edges relate to the transitions between the stages. A path from the source of the DAG to the sink defines a particular *situation* in the address translation. Such situations can be referred from high-level descriptions of *test programs* (TPs), so-called *test templates* (TTs). The latter are processed by Genesys-Pro [1], which formulates constraints on instruction operands, solves them and transforms the solutions into the instruction sequences. The major advantage of the approach is the use of the highly developed languages for modeling MMUs and describing TTs. A possible disadvantage is that the tool seems not to be able to automatically extract MMU-related *dependencies* between instructions.

In [10], the Java language coupled with a special library is used to model MMUs. As in DeepTrans, the situations correspond to the paths in the DAG describing the MMU. For example,  $\{TLB(va).hit, TLB(va).entry.V, \neg L1(pa).hit\}$ : there is a hit in the *translation lookaside buffer* (TLB); the matched entry is valid; there occurs a miss in the *first-level cache* (L1). In addition, the approach provides facilities for specifying MMU-related dependencies between instructions. For example,  $\{TLB \mapsto \neg tagEqual, L1 \mapsto indexEqual\}$ : instructions access different TLB entries; data are mapped onto the same set of L1. TTs are constructed automatically

by combining situations and dependencies for short sequences of instructions. Building TTs and creating TPs is done by MicroTESK (version 1) [5]. The strength of the approach is systematic TT enumeration that takes into consideration instruction execution paths as well as dependencies between instructions. The principal weakness is underdeveloped specification facilities.

### III. MICROTesk FRAMEWORK

MicroTESK (version 2.3 or higher) [11] combines the advantages of the approaches presented in [9] and [10]. The tool inputs are ISA specifications in nML [6], MMU specifications in MMUSL (MMU Specification Language) and TTs in Ruby [12]. The basic principles of MicroTESK are close to ones implemented in Genesys-Pro [1]. The specifications are analyzed to extract *testing knowledge* (situations and dependencies), which is used to generate TPs from the given TTs as well as to systematically enumerate TTs. More information on the tool can be found in [13] and [14]. Here we provide a brief introduction to MicroTESK by the example of an MIPS64 MMU [8].

#### A. ISA Specifications

ISA specifications include definitions of *data types*, *constants*, *registers*, *access modes*, *memories* and *instructions*. Here comes an example (a fragment of the MIPS64 specification), where there are listed three data types, BYTE, SHORT and DWORD.

```
type BYTE = card(8) // 8-bit Bit Vectors (unsigned)
type SHORT = int(16) // 16-bit Bit Vectors (signed)
type DWORD = card(64) // 64-bit Bit Vectors (unsigned)
```

Registers of the same type are grouped into arrays. Register access logic is encapsulated in so-called *modes*, which, besides other things, define *assembly format (syntax)* and *binary encoding (image)* of the registers. The following example declares an array GPR, consisting of thirty two 64-bit registers, designates a *stack pointer* alias SP = GPR[29], and defines a mode REG aimed at accessing those registers.

```
reg GPR[32, DWORD] // Array of 32 DWORD Registers
reg SP[DWORD] alias = GPR[29] // Stack Pointer Alias

mode REG(i: card(5)) = GPR[i] // One-to-One Correspondence
  syntax = format("r%d", i) // Assembly Format (e.g., r13)
  image = format("%5s", i) // Binary Encoding
  number = i // Register Number (custom)
```

Like a group of registers, a memory unit is represented as a plain array. In the example below, an array MEM is interpreted as a *physical memory* comprised of  $2^{36}$  bytes. *Virtual memory* issues such as address translation, caching, and the like are specified separately with the use of a dedicated language MMUSL (see the next section).

```
mem MEM[2 ** 36, BYTE] // Physical Memory Array
```

The attributes of instructions include **syntax**, **image** and **action**. Actions of load and store instructions are described in an intuitive manner by reading or writing data from or to the array representing the physical memory. Here is a specification

of the *Load Byte* instruction (LB), which derives an address from a base register (base) with given offset (offset), loads a byte from the memory, and writes it to a register (rt).

```
op LB(rt: REG, offset: SHORT, base: REG)
  syntax = format("lb%s,%d(%s)",
    rt.syntax, offset, base.syntax)
  image = format("100000%5s%5s%16s",
    base.image, rt.image, offset)
  action = {
    rt = MEM[base + offset];
  }
```

Notwithstanding MEM is interpreted as the physical memory, it is accessed through virtual addresses – an access triggers the address translation mechanisms and other MMU logic.

#### B. MMU Specifications

Being rather simple, nML does not have adequate facilities to describe MMUs. For this purpose, a special MMUSL language is used. MMU specifications include *address types*, *memory segments*, *buffers*, and *control logic* for handling loads and stores. In the following example, address type, VA, is declared. It is a structure with single field – address itself.

```
address VA(vaddress : 64) // Virtual Address
```

A memory segment is considered as a mapping from a set of addresses of some type to a set of addresses of another type. An example given below defines a segment XKPHYS that maps a VA of the given set (**range**) to the physical address (PA). The segment performs flat translation with no use of TLBs and tables (**read**).

```
segment XKPHYS(va: VA) = (pa: PA)
  range = (0x8000000000000000, 0xbfffffffffffffff)
  read = {
    pa.paddress = va.vaddress<35..0>;
    pa.cca = va.vaddress<61..59>;
  }
```

Buffers (TLBs, cache units, page tables, etc.) are specified with the following parameters: the *associativity (ways)*, the *number of sets (sets)*, the *entry format (entry)*, the *index calculation function (index)*, the *tag calculation function (tag)* and the *data eviction policy (policy)*. Their meaning passes current among microprocessors designers. Here comes a sample description of TLB. It is accessed by VAs. The keyword **register** means that the buffer is *mapped to the registers* and can be accessed from the ISA specifications.

```
register buffer TLB(va: VA)
  sets = 1 // Fully Associative Buffer
  ways = 64
  entry = (R: 2, VPN2: 27, ASID: 8, PageMask: 16, G: 1, ...)
  tag = va<39..13>
  policy = NONE // Non-replaceable Buffer
```

Processing of memory access instructions is specified by requesting the segments and buffers. The syntax is similar to nML though allows using such constructs as B(A).**hit** (the buffer B contains an entry for the address A), E = B(A) (the entry for the address A is read from the buffer B and assigned to E), B(A) = E (the entry E for the address A is written to the buffer B), and the like. Here is a fragment of the MIPS64 MMU specification. It contains two attributes,

**read** and **write**, which, respectively, define logic of loads and stores.

```
mmu MMU (va: VA) = (data: DATA_SIZE)
var pa: PA;
var isCached: 1;
var line: DATA_SIZE;
var l1Entry: L1.entry;
read = {
  pa = TranslateAddress(va);
  isCached = IsCached(pa.cca);
  if isCached == 1 then
    if l1(pa).hit then // L1 Cache Access
      l1Entry = l1(pa);
      line = l1Entry.DATA;
    else
      ...
      line = MEM(pa);
      l1Entry.TAG = pa.value<...>; // L1 Cache Update
      l1Entry.DATA = line;
      l1(pa) = l1Entry;
    endif;
  else
    line = MEM(pa);
  endif;
  data = line;
}
write = { ... }
```

### C. TPG Approach

The MicroTESK TPG approach is based on TTs written in Ruby [12]. In general terms, the process is as follows [14]. A TT describing a microprocessor verification scenario is given to MicroTESK. The tool processes the TT and builds a series of *symbolic TPs*, where abstract *situations* and *dependencies* (often in the form of *constraints*) are used instead of specific values. Each symbolic TP is instantiated with appropriate *test data (TD)*. The resultant TP is supplemented with *preparation code* that initializes the registers, the buffers, and the memory.

TTs are allowed to use modes and instructions defined in the specifications as well as special TPG constructs (*blocks*, *situations*, etc.) [14]. More technically, a TT is a subclass of the Template base class provided by the MicroTESK library. In the example below, MmuTemplate is a subclass of Mips64BaseTemplate, which, in turn, is a subclass of Template. The entry point is run. This method declares a *block* of two instructions, LD and SD, to be processed with the dedicated memory engine. The situation access guides TPG by specifying constraints and biases for the MMU variables and buffers. The denotation  $\text{reg}(\_)$  means any instance of the mode REG, i.e. any GPR register.

```
class MmuTemplate < Mips64BaseTemplate
  def run # Test Template Entry Point
    block(:engine => "memory", ...) {
      ld reg(_), 0x0, reg(_) # Load Double Word Instruction
      do situation("access", hit("L1"), ...) end
      sd reg(_), 0x0, reg(_) # Store Double Word Instruction
    }
  end
end
```

Let us consider how TPG for MMUs is organized. Parsing specifications results in two entities: an *interpreter*, which is a part of the *instruction set simulator (ISS)*, and a *symbolic representation* in the form of a labeled DAG. The DAG is traversed, and all possible *execution paths* are extracted. An execution path describes processing of a single memory

request and finishes either with a *memory access* or with an *exception* (alignment fault, TLB refill event, etc.). Paths are composed of transitions. Each transition is supplied with a *guard*, i.e. a condition that enables the transition, and an *action* to be performed; it can also be labeled with a *buffer* being used in the guarded action. Here is a fragment of the execution path in MMU (see above) represented in a hypothetical language.

```
path PATH(va: VA) = (data: 64)
transition {
  guard = TRUE
  action = {} // Go to TranslateAddress(va)
} ...
transition {
  guard = l1(pa).hit
  action = { l1Entry = l1(pa); line = l1Entry.DATA; }
  buffer = L1
} ...
```

Given two execution paths, the tool can extract possible *dependencies* between them. A dependency is a map from the set of buffers common for the given paths to the set of *conflict types*. More formally, let  $p_1$  and  $p_2$  be execution paths,  $C$  be a non-empty set of conflict types, and  $B(p)$  be the set of buffers used in a path  $p$ . A dependency between  $p_1$  and  $p_2$  is a map  $d : B(p_1) \cap B(p_2) \rightarrow C$ . The set  $C$  is supposed to include the following elements and their negations:

- *indexEqual* – access to the same set of the buffer:
  - *tagEqual* – access to the same entry of the buffer;
  - *tagEvicted* – access to the recently evicted entry.

Given a TT, symbolic TPs are systematically enumerated. The main, but not the only, approach supported by MicroTESK is *combinatorial generation*. Symbolic TPs are constructed by selecting all relevant execution paths for the TT's instructions and producing all satisfiable dependencies for each combination of the paths. To avoid combinatorial explosion, special *heuristics* are used, including factorization of the paths and limitation of the depth of the dependencies. Among them, a *buffer-event factorization* is frequently used. Let  $p$  be a path, and  $\text{event}_p : B(p) \rightarrow \{\text{hit}, \text{miss}\}$  be the induced map of the buffers to the events. Two paths,  $p_1$  and  $p_2$ , are *equivalent*, if  $B(p_1) = B(p_2)$  and for each  $b \in B(p_1)$ ,  $\text{event}_{p_1}(b) = \text{event}_{p_2}(b)$  holds. During TPG, the equivalence classes are enumerated, while their representatives are randomized.

Symbolic TP is a pair  $\langle \{p_i\}_{i=1}^n, \{d_{ij}\}_{i,j=1(i < j)}^n \rangle$ , where  $p_i$  is an execution path, and  $d_{ij}$  is a dependency between  $p_i$  and  $p_j$ . To produce a TP from a symbolic TP, appropriate TD are required, including addresses of the instructions, entries of the buffers being accessed (except *replaceable* ones, such as caches) and sequences of addresses to be used to load or evict data to or from the replaceable buffers. Formally, TD are a tuple  $\langle \{\text{addr}_i\}_{i=1}^n, \{\text{entry}_i\}_{i=1}^n, \text{load}, \text{evict} \rangle$ , where  $\text{addr}_i(a)$  is an address of the type  $a$  used in the path  $p_i$ ,  $\text{entry}_i(b)$  is an entry of the buffer  $b$  accessed by the path  $p_i$ ,  $\text{load}(b, s)$  is a sequence of addresses to load data to the set  $s$  of the buffer  $b$  and, finally,  $\text{evict}(b, s)$  is a sequence of addresses to evict data from the set  $s$  of the buffer  $b$ .

Here is an approximation of the TD generation algorithm implemented in MicroTESK's memory engine. The following denotations are used:  $d_j(b, c)$  is the minimal  $i$ , such that

$1 \leq i < j$  and  $d_{ij}(b) = c$ , or a special value  $\epsilon \notin \mathbb{N}$  if there are no such  $i$ ;  $addr_j(b)$  is equivalent to  $addr_j(a_b)$ , where  $a_b$  is the address type of the buffer  $b$ ;  $tag_b(addr)$  and  $index_b(addr)$  are, respectively, the tag and the index extracted from  $addr$  by using the corresponding functions of the buffer  $b$ ;  $newAddr_b(tag, index, \dots)$  is an address constructed from  $tag$ ,  $index$ , and, probably, some other information;  $newEntry_b(id, index)$  is an empty entry of the buffer  $b$  with specified  $id$  and  $index$ ; given a buffer  $b$ , its state  $s$ , and  $index$ ,  $victim_b(s, index)$  is a tag to be evicted.

---

#### Algorithm 1 Generator

---

**Input:** Symbolic TP:  $\langle \{p_i\}_{i=1}^n, \{d_{ij}\}_{i,j=1(i < j)}^n \rangle$   
**Output:** Generated TD:  $\langle \{addr_i\}_{i=1}^n, \{entry_i\}_{i=1}^n, load, evict \rangle$

```

for all  $j \in \{1, \dots, n\}$  do
   $addr_j \leftarrow Solver.constructAddresses(p_j)$  ▷ Construct Addresses
  for all  $b \in B(p_j)$  do ▷ Process Tag/Index Equalities
    if  $d_j(b, tagEqual) \neq \epsilon$  then
       $i \leftarrow d_j(b, tagEqual)$ 
       $addr_j(b) \leftarrow newAddr_b(tag_b(addr_i(b)), index_b(addr_j(b)), \dots)$ 
    else if  $d_j(b, indexEqual) \neq \epsilon$  then
       $i \leftarrow d_j(b, indexEqual)$ 
       $tag_{new} \leftarrow Allocator.allocTag(b, index_b(addr_i(b)))$ 
       $addr_j(b) \leftarrow newAddr_b(tag_{new}, index_b(addr_i(b)), \dots)$ 
    end if
  end for
  for all  $b \in B(p_j)$  do ▷ Process Hits and Misses
    if  $b.policy \neq none$  then ▷ Replaceable Buffer
      if  $event_{p_j}(b) = hit$  then
         $load(b, index) \leftarrow load(b, index) \cdot \{addr_j(b)\}$ 
      else
        for all  $k \in \{1, \dots, b.ways\}$  do
           $tag_{new} \leftarrow Allocator.allocTag(b, index_b(addr_j(b)))$ 
           $addr_{evict} \leftarrow newAddr_b(tag_{new}, index_b(addr_j(b)), \dots)$ 
           $evict(b, index) \leftarrow evict(b, index) \cdot \{addr_{evict}\}$ 
        end for
      end if
    else ▷ Non-Replaceable Buffer
      if  $event_{p_j}(b) = hit$  then
        if  $d_j(b, tagEqual) \neq \epsilon$  then
           $i \leftarrow d_j(b, tagEqual)$ 
           $entry_j(b) \leftarrow entry_i(b)$ 
        else
           $id_{new} \leftarrow Allocator.allocEntryId(b, index_b(addr_j(b)))$ 
           $entry_j(b) \leftarrow newEntry_b(id_{new}, index_b(addr_j(b)))$ 
        end if
      end if
    end if
  end for
   $state \leftarrow Interpreter.observeState()$  ▷ Process Data Evictions
   $loads \leftarrow Loader.prepareLoads(load, evict)$ 
   $state \leftarrow Interpreter.execMmu(loads, state)$ 
  for all  $j \in \{1, \dots, n\}$  do
    for all  $b \in B(p_j, a)$  do
      if  $d_j(b, tagReplace) \neq \epsilon$  then
         $i \leftarrow d_j(b, tagReplace)$ 
         $addr_j(b) \leftarrow newAddr_b(Tag_{evict}(b, p_i), index_b(addr_j(b)), \dots)$ 
      end if
      if  $event_{p_j}(b) = miss$  then
         $Tag_{evict}(b, p_j) \leftarrow victim_b(state, index_b(addr_j(b)))$ 
      end if
       $state \leftarrow Interpreter.execBuffer(b, \{addr_j(b)\}, state)$ 
    end for
  end for
  for all  $j \in \{1, \dots, n\}$  do ▷ Construct Entries
     $entry_j \leftarrow Solver.constructEntries(p_j)$ 
  end for

```

---

*Generator* exploits several auxiliary components: *Solver*, *Allocator*, *Interpreter* and *Loader*. *Solver* performs symbolic execution of a given path and constructs required entities (addresses, entries, etc.) by calling constraint solvers. Interface with solvers is provided by Fortress library [15]. It supports

*SMT solvers*, such as Z3 [16] and CVC4 [17], as well as *in-house solvers* aimed at particular tasks. *Allocator* chooses buffer indices, tags and other address fields taking into account user-defined constraints (e.g., forbidden memory regions). The default strategy is to allocate a new index or a new tag for a given index on every request. This allows avoiding undesirable dependencies between instructions. *Interpreter* simulates accesses to buffers and predicts data evictions. The results of the predictions are used to satisfy *tagEvicted* conflicts. *Loader* prepares a sequence of accesses so as to fulfil *hit* and *miss* requirements. The default strategy is as follows. Buffers are handled in reverse order; for every buffer  $b$  and every set  $s$ ,  $evict(b, s)$  and  $load(b, s)$  are added to the sequence.

Finally, TD are transformed to the ISA-specific preparation code. For this job, the tool needs to know what instructions have to be used to set up addresses and entries. Such information is provided in TTs in the form of so-called *preparators*. Technically, a preparator is a piece of code that defines a sequence of instructions to reach a certain goal. Given a register type (to be more precise, an access mode), there usually exists a family of preparators differing in patterns of loaded values. For example, `Mips64BaseTemplate` contains the following preparator for loading a 32-bit value into an GPR register via the mode REG.

```

preparator(:target => "REG", :mask => "00000000XXXXXXXX") {
  ori target, target, value(16, 31)
  dsll target, target, 16
  ori target, r0, value(0, 15)
}

```

For each buffer, there should be a preparator to write an entry into it. A preparator for MIPS64 DTLB is given below.

```

buffer_preparator(:target => 'DTLB') {
  ori t0, r0, address(48, 63)
  dsll t0, t0, 16
  ori t0, t0, address(32, 47)
  dsll t0, t0, 16
  ori t0, t0, address(16, 31)
  dsll t0, t0, 16
  ori t0, t0, address(0, 15)
  lb t0, 0, t0
}

```

## IV. MIPS64 MMU CASE STUDY

The most challenging part of creating a specification-based TPG tool for a microprocessor is MMU specification. Speaking of MIPS64, the following is defined [8]: fixed set of address spaces, TLB entry format and address translation procedure. Moreover, the system under test uses two-level write-through cache.

MicroTESK's MMUSL allowed to specify MIPS64 in quite a compact way (approximately 220 lines of code). The specifications involve the TLB (JTLB and DTLB), two-level cache memory buffers (L1 and L2) and fixed memory segments (kseg0, kseg1, xkphys, useg). On the base of the ISA [18] and MMU [8] specifications, 18 memory access instructions are defined with additional instructions to read from and write to TLB. Description of a single instruction makes up approximately 10 lines of nML code on average.

TABLE I  
COMPLEXITY OF MIPS64 MMU SPECIFICATIONS

CHARACTERISTIC	MINIMUM	MAXIMUM	AVERAGE
Number of Transitions in an Execution Path	7	52	38
Number of Variables in a Path Formula	3	76	49
Number of Execution Paths of an Instruction	76		

Table I contains numeric data on MIPS64 MMU execution path complexity. While the complexity is relatively low (average path consists of less than 40 transitions and comprises under 50 variables) it makes exhaustive enumeration of symbolic TPs reasonable only for very short TTs. In more complicated cases heuristics become of crucial importance. E.g., the buffer-event factorization gives only 9 path equivalence classes, enabling systematic enumeration of longer sequences of memory accesses. Generation of even more complicated TPs is done with the help of the constrained random generation. This requires verification engineers to explicate their knowledge in the form of constraints and biases.

This is an ongoing project, and some useful information, such as test coverage, is not available at the moment. Though it is worth considering the lessons learned. We found it convenient to use domain-specific languages (DSLs) for specifying ISAs and MMUs. The use of DSLs, first, eases extraction of testing knowledge and, second, simplifies learning of the TPG tool. On the other hand, it seems that dynamic programming languages, such as Ruby and Python, suit well for describing TTs. Such languages can be easily extended with TPG constructs. Our negative experience is mostly connected with low performance of the tool. Constraint solving needs to be optimized. As the authors of [19], we believe that specialized solvers would help.

## V. CONCLUSION

TPG is a widely-accepted approach to microprocessor verification, including, in particular, MMU verification. State-of-the-art MMUs are extremely complex devices comprising multi-level address translation and caching. Naïve approaches to automated TPG for MMUs – meaning, first of all, random generation techniques – are highly improbable to reach high level of test coverage in reasonable time. Specification-based TPG, in our opinion, is one of the most promising directions in the area. Since 1990s, it has been successfully applied to microprocessor testing and verification, e.g., in IBM [1], and it continues to evolve.

The MicroTESK team [5] contributes its mite to the evolution of the specification-based approach. Our goal is to create an open-source, extensible and reconfigurable TPG framework [13], [14]. Different versions of MicroTESK, including the one described in [10], have been applied to several industrial microprocessors and allowed to reveal a large number

of critical bugs, which had not been detected by randomly generated TPs.

The proposed solution is based on ISA specifications in nML [6] and MMU specifications in MMUSL. ISA specifications formally describe microprocessor instructions, while MMU specifications define memory segments and buffers. MicroTESK is able to automatically extract testing knowledge from the specifications and to exploit it for TPG. TTs are created with the help of Ruby [12]. To generate TD, symbolic execution and constraint solving techniques are intensively used.

The work is still in progress, and a number of things need to be done. The most priority task is a performance optimization of the constraint solving. Another task is to extend the approach to multicore designs and multiprocessor systems. The main challenge here is to create a unified technology that would include formal verification of cache coherence protocols, unit-level verification of MMUs, and system-level TPG.

## REFERENCES

- [1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimón, M. Vinov, A. Ziv. *Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification*. Design & Test of Computers, 2004. pp. 84–93.
- [2] R.L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002. 224 p.
- [3] T. Li, D. Zhu, Y. Guo, G. Liu, S. Li. *MA2TG: A Functional Test Program Generator for Microprocessor Verification*. Euromicro Conference on Digital System Design, 2005. pp. 176–183.
- [4] A. Kamkin, A. Tatarnikov. *MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors*. Spring/Summer Young Researchers Colloquium on Software Engineering, 2012. pp. 64–69.
- [5] *MicroTESK page* – <http://forge.ispras.ru/projects/microtesk>
- [6] M. Freericks. *The nML Machine Description Formalism*. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.
- [7] MIPS64 Architecture For Programmers. Volume 3: MIPS64/microMIPS64 Privileged Resource Architecture. Revision 6.03. MIPS Technologies Inc. – 2015. – 368 p.
- [8] A. Adir, L. Fournier, Y. Katz, A. Koyfman. *DeepTrans – Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms*. High-Level Design Validation and Test Workshop, 2006. pp. 102–110.
- [9] D. Vorobyev, A. Kamkin. *Generatsiya testovykh programm dlya podsystemy upravleniya pamyat'yu mikroprotsessora [Test Program Generation for Memory Management Units of Microprocessors]*. Trudy ISP RAN, 17, 2009, pp. 119–132 (in Russian).
- [10] A. Kamkin, A. Protsenko, A. Tatarnikov. *An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms*. Trudy ISP RAN, 27(3), 2015. pp. 125–138.
- [11] *Ruby site* – <http://www.ruby-lang.org>
- [12] A. Kamkin, E. Kornychin, D. Vorobyev. *Reconfigurable Model-Based Test Program Generator for Microprocessors*. International Conference on Software Testing, Verification and Validation Workshops, 2011. pp. 47–54.
- [13] A.S. Kamkin, T.I. Sergeeva, S.A. Smolov, A.D. Tatarnikov, M.M. Chupilko. *Extensible Environment for Test Program Generation for Microprocessors*. Programming and Computer Software, 40(1), 2014. pp. 1–9.
- [14] *Fortress page* – <http://forge.ispras.ru/projects/solver-api>
- [15] *Z3 page* – <http://github.com/Z3Prover/z3>
- [16] *CVC4 site* – <http://cvc4.cs.nyu.edu>
- [17] MIPS64 Architecture For Programmers. Volume 2: The MIPS64 Instruction Set Reference Manual. Revision 6.04. MIPS Technologies Inc. – 2015. – 551 p.
- [18] Y. Naveh, M. Rimón, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, G. Shurek. *Constraint-Based Random Stimuli Generation for Hardware Verification*. AI Magazine, 28(3), 2007. pp. 13–30.

# Approaches to Stand-alone Verification of Multicore Microprocessor Caches

Mikhail Petrochenkov<sup>1</sup>, Irina Stotland<sup>2</sup>, Ruslan Mushtakov<sup>3</sup>

Department of Verification and Modeling

MCST

Moscow, Russia

[petroch\\_m@mcst.ru](mailto:petroch_m@mcst.ru), [stotl\\_i@mcst.ru](mailto:stotl_i@mcst.ru), [mushtakov\\_r@mcst.ru](mailto:mushtakov_r@mcst.ru)<sup>3</sup>

**Abstract**—The paper presents an overview of approaches used in verifying correctness of caches of multicore microprocessors. Approaches for designing a test system, generating valid stimuli and checking correctness of the device behaviour are introduced. Some novel methods to verify functionally nondeterministic devices are described. Additionally, we describe how the test systems for devices, that support out of order execution, could be designed. In conclusion we provide a case study of using these methods to verify caches of microprocessors with “Elbrus” architecture and “Sparc v” architecture.

**Keywords**—multicore microprocessor, cache memory, out-of-order execution, test system, indeterministic behaviour, model-based verification, stand-alone verification.

## I. INTRODUCTION

The key feature of modern microprocessor architecture is multicoreness - combining several computational cores on a single system on chip (SOC). To reduce time needed to access RAM, device can incorporate several levels of cache hierarchy. Access to smaller caches can be executed faster than access to larger caches of the next level of hierarchy. Caches can keep data for a single computational core or serve as a data storage for several of them at the same time. Memory subsystem of a multicore microprocessor must maintain coherence of the memory. Task of maintaining correct state of memory is usually solved by implementing cache coherence protocol that defines a set of data states and actions on transitions between states in cache[1]. To optimize design and implementation of coherency protocol, caches can include local directory – device that keeps information on states of data in different components of memory subsystem. Sufficient complexity of protocols and their implementations in multilevel memory subsystems can lead to hard to find errors. To ensure robustness of a microprocessor, one must thoroughly verify its memory subsystem.

Several works present approaches used in formal verification of cache coherence protocols. To check device implementation functional verification is used. One of the approaches to microprocessor verification is system verification - execution of test programs on microprocessor model and on reference implementation of its instruction set, and comparison between them. It should be noted that caches are often invisible from the point of view of a programmer. That is why design of programs capable of sufficient verification of a microprocessor caches is a complex task. To

ensure adequate level of verification stand-alone verification is used. This paper addresses the problem of stand-alone verification of microprocessor caches of different levels.

The rest of the paper is organized as follows. Section 2 reviews the existing techniques for designing test oracles. Section 3 suggests an approach to the problem. Section 4 describes a case study on using the suggested approach in an industrial setting. Section 5 concludes the paper

## II. COMMON VIEW ON STAND-ALONE VERIFICATION OF MICROPROCESSOR CACHES

The object of stand-alone verification is model of verifying device implemented in hardware description language (usually, Verilog or VHDL). It defines the behavior of the device on a register transfer level (RTL). The device specification defines a set of stimuli and reactions based on the state of the device. To check correctness of the device it is included in a test system - a program that generates test stimuli, checks validity of reactions and determines verification quality. Based on its functions test system can be divided into separate modules - stimulus generator and correctness checking module (test oracle). Methods of estimation of verification quality are similar to that of other devices. Information on code coverage is used to identify unimplemented test scenarios and refine stimulus generator. This approach is called coverage driven constrained random verification. In addition the method described above, different approaches were presented: using formal approaches to ensure full coverage of cache coherence protocol implementation in the device[3].

Cache behaviour exhibits a set of properties that should be considered while designing a test system for verification of the device.

- Transactions in the system can be separated into three groups: primary requests, secondary requests and reactions.
- A device implements a part of cache coherence protocol.
- A device works independently with different cache lines - areas of memory of fixed size.
- Requests that work with the same cache line are serialized.

- Device implements data eviction mechanism and protocol to determine victim line (usually some variant of LRU).

Using these properties of the device under testing while designing a test system could lead to simplified structure of the system and improved performance.

### III. TEST STIMULI GENERATION

#### A. The general approach

Test stimuli are usually generated on a more abstract level than register transfers and interface signals. Based on the logical and functional similarity, groups of device ports are combined into interfaces. Interfaces are used to transfer transaction level packets[7]. To transform packets between different representations on signal and transaction level, serializer and deserializer modules are implemented[2].

Test system should generate stimuli similar to that in a real system. Should be noted that primary requests in real microprocessor are consequences of some memory access operation (loading, storing data, eviction, prefetch, atomic swap, etc). Secondary requests are answers for reaction packets from the device. It is usually convenient to use only a sequence of primary requests as a test sequence, and generate secondary requests automatically in corresponding modules. Properties of secondary requests could be changed based on secondary request generation modules configuration.

In the test system interfaces are combined into groups that represent working with some devices. Test system should simulate the state of the devices to generate correct responses from it.

#### B. Generation of primary requests for caches with out-of-order execution

Properties of the devices that support out-of-order execution should be considered while designing stimulus generator:

- Order of primary request can be different from the order of memory accesses in initial program.
- Primary request could be separated into several messages accepted at separate times. Messages for one primary request are identified by common value of tag field.
- Request canceling mechanism is present.

To support out-of-order execution of memory access requests in a cache common approach was augmented. The module responsible for transfer of primary requests was replaced with high-level module that includes components working with interfaces of primary request parts. Order of request for that module is identical to that of a test program, and reordering of request parts is executed based on module settings.

### IV. CORRECTNESS CHECKING

One of the ways to check correctness of the device behavior is a comparison with a reference model, implemented

either in general purpose programming language (C, C++) or in specialized hardware verification language (SystemVerilog, "e", Vera). If test stimuli are the same, difference in model and device reactions means an error somewhere in the system[2]. Reference models could be cycle-accurate or functional (behavioral). To implement the former, behavior of the device must be specified on a register transfer level. Behavior of caches usually defined on a higher level of abstraction, because cache is not an essential part of computational pipeline of a microprocessor. A cache is not a subject of strict temporal requirements. To verify caches functional models working on transaction level are implemented.

#### A. Checking of indeterministic caches

If one wants to develop functional model of cache, its specification must have property of transaction level indeterminism. That is, identical transaction level traces of stimuli (a set of register transfer level traces is mapped into this single transaction level trace) must cause identical transactional reaction trace. It should be noted that caches often include a set of components (eviction arbiter, primary request arbiter serving different requesters), that do not hold that property. That is, different register transfer level traces that are mapped into single transaction level trace could lead to different reaction traces. There are several methods to check behaviour of indeterministic devices.

1) *"Gray box" method*: one of the ways to solve aforementioned problem is to replace usual "black box" method of device verification. That is, we should not consider only external interfaces of the device while analysing its behaviour. To determine which variant of behaviour has happened in the cache one could use "hints" from the implementation. To use this approach, a set of internal interfaces and signals is defined and its behaviour is specified. This interfaces must be chosen in a way that information on their state could be used to eliminate indeterminism. In general, in caches such signals are results of primary request arbitration and interfaces of finite automata of cache eviction mechanism. Additionally, that information can be used in request generator and for the estimation of verification quality. This method is usually easy to implement. Drawbacks of this methods are additional requirements for specification and reliance on interfaces that could also exhibit erroneous behaviour.

2) *Dynamic refinement of behavioural model*: Another approach is to create additional instances of model for each variant of behaviour in case of nondeterministic choice in the device[4]. Each reaction is checked against every spawned device model. If reaction is impossible for one variation of behaviour, then it is removed from set. If set of possible states after some reaction becomes empty, the system must return error. In general, this approach may cause exponential growth of number of states with each consecutive choice. But for caches this approach could be implemented efficiently, because of several properties of caches: serialization of requests and cache line independence. Information on which indeterministic choice was made in the device (for use in



request generator or for verification quality estimation) could also be extracted from reactions. Strong points of that approach compared to “gray box” method is elimination of reliance on implementation details of the device. Drawback is additional complexity of implementation.

3) *Assertions*: Test stimulus generators simulate the behaviour of the device under test. It also should be noted that interaction between the device and its environment must adhere to some protocol. Based on that protocol, one can include functional requirements of protocols as an assertions in the generator. Then, violation of an assertion represents signals an error. Usage of assertions is an effective method of detection of a broad class of errors. In addition to assertions that are common for all memory subsystem devices, several cache-specific assertions could be included. They represent invariants of cache coherence protocol. To check this invariants, coherence of states of a single cache line is analyzed in all parts of test system after each change.

### B. Checking caches with out-of-order execution

For caches supported out of order request execution, it exhibits properties of limited indeterminism. That is memory access request are received in the device in multiple parts from several interfaces, with different unspecified timing characteristics. On the other side there is “reference” order of memory access operations, present in original test program. If out of order execution introduces error to the canonical order, device must be cleaned and erroneous transactions must be restarted. Results of operations that completed successfully are deterministic. Based on these properties of the device, two modes of operation we implemented:

- “Ignore the cancelled transactions” mode.
- Strict checking mode

In the first mode result checking is delayed until the moment of its full completion. If completion was unsuccessful, checks are not made. In the strict mode, approaches similar to dynamic refinement of model could be used. Set of possible device states is maintained, and it is augmented with each stimulus and reaction. Number of possible states is limited by the number of simultaneously executed out of order requests. Shortcomings of the first mode are delay between erroneous

SPARC microprocessor (8 cores, frequency - 2GHz) is the first MCST microprocessor that supports out-of-order execution of requests. L1DC serves as a data storage for a single core and support receiving of requests in order that is not the same as in the source program. Simplified model of computational core with reorder buffer (ROB) serves as a part of a stimulus generator in test system. The generator also contains the model of level two data cache (data cache for four cores) and models of other three level one caches to simulate interactions between cores and analyze the state of the system (pic.1).

transaction and the execution of actual checking and reduction of the set of errors that could be detected (for example unnecessary cancel of request will not be detected). On the other hand, implementing that mode is much simpler task, so verification could be started sooner.

## V. CASE STUDY

Methods described above were used in the process of verifying the L2-cache[6] and the L3-cache[4] of the microprocessor with “Elbrus” architecture and level one data cache of a microprocessor with “SPARC-V9” architecture. Caches of “Elbrus” microprocessor are part of a system on chip (SOC) with 8 computational cores. Each core has level one and level two (2MB) cache. Level three cache (16 MB) is shared between all cores. Size of level one cache (L1DC) of SPARC microprocessor is 32KiB, device support out of order execution of memory access operations. The test system structure for L1DC is presented in figure 1.

Test stimulus generator was developed to verify the L3-cache of the “Elbrus” microprocessor[5]. It is based on simplified model of microprocessor core with the L2-cache and the model of system commutator that simulates work in multiprocessor environment. If multiple cores request access to a single cache line, then order of their execution is unspecified and defined by device microarchitecture. Internal structure of a cache is also a subject of change, due to changes to requirements of physical design. To verify the device approach based on dynamic refinement of behavioral model was chosen. To supplement that approach, a set of assertions were implemented in stimulus generator to check validity of system state. Using that approach allowed to use the same test system with minimal alteration for the next iteration of the “Elbrus” microprocessor

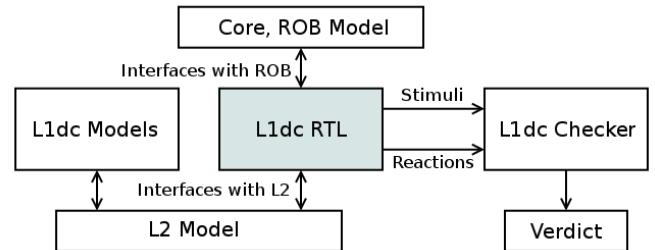


Fig. 1. Principal sturcture of test sytem for level one data cache of SPARC microprocessor.

## VI. CONCLUSION

Approaches, described in this article were used while verifying caches of microprocessors developed by MCST. Stand alone verification allowed finding several errors in different caches. The intermediate results of application introduced approaches in multicore microprocessor caches verification if presented in table 1.

TABLE I. APPLICATION RESULTS

	Verified caches		
	<i>L2-cache "Elbrus"</i>	<i>L3-cache "Elbrus"</i>	<i>L1 data cache "SPARC-V9"</i>
Number of bugs	3	4	8

Using of aforementioned methods while developing test systems allowed increasing quality of verification: allowed to achieve higher code coverage and reduced the amount of false-positive results. Approaches could be used to verify other caches of different multicore microprocessors regardless of its architectures.

Our future research is connected with improving the error diagnostics and localization of found bugs.

## REFERENCES

- [1] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011. 195 p.
- [2] Stotland I., Lagutin A. Primenenye etalonnykh sobytijnykh modeley dlya avtonomnoy verifikatsii modulei microprocessorov [Using stand alone behavioural models to verify microprocessor components]. Voprosy radioelektroniki, seriya EVT, 2014, 3, p. 17-27.
- [3] Kamkin A., Petrochenkov M. Sistema podderzhki verifikatsii kogerentnosti s ispol'zovaniem formal'nykh metodov [A system to support formal methods-based verification of coherence protocol implementations]. Voprosy radioelektroniki, seriya EVT, 2014, 3, p. 27-38.
- [4] Kamkin A., Petrochenkov M. A Model-Based Approach to Design Test Oracles for Memory Subsystems of Multicore Multiprocessors. Trudy ISP RAN, vol. 27, 3, p 149-157.
- [5] Kozhin A., Kozhin E., Kostenko V., Lavrov A. Kesh kogerentnosti mikroprotssora «El'brus-4S+» [L3 cache and cache coherence support in «Elbrus-4C+» microprocessor]. Voprosy radioelektroniki, seriya EVT, 2013, 3, p. 26-38.
- [6] Stotland I., Kutsevol V., Meshkov A. Problemy funktsionalnoy verifikatsii kesh pamyati vtorogo urovnya mirkoprocesorov arkhitekturi "Elbrus" [Challenges of functional verification of level two cache of "Elbrus" microprocessor]. Voprosy radioelektroniki, seriya EVT, 2015, 1, p. 76-84.
- [7] TLM-2.0.1. TLM Transaction-Level Modeling Library. URL: <http://www.accellera.org/downloads/standards/systemc> (20.12.2015).
- [8] The SPARC architecture manual:version 9 / SPARC International, Inc. ; David L. Weaver, Tom Germond, editors.: PTR Prentice Hall, c1994. xxi, 357p. ISBN 0-13-099227-5

# Checking Parameterized PROMELA Models of Cache Coherence Protocols

V.S. Burenkov<sup>1</sup>, A.S. Kamkin<sup>2</sup>

<sup>1</sup>JSC MCST, burenkov\_v@mcst.ru

<sup>2</sup>ISP RAS, kamkin@ispras.ru

**Abstract**—This paper introduces a method for scalable functional verification of cache coherence protocols described in the PROMELA language. Scalability means that verification efforts do not depend on the model size (i.e. the number of processors in the system under verification). The method is comprised of three main steps. First, a PROMELA specification written for a certain configuration of the system under verification is generalized to the specification being parameterized with the number of processors (to do it, some assumptions on the protocol are used as well as simple induction rules). Second, the parameterized specification is abstracted from the number of processors (it is done by syntax transformation of the specification). Finally, the abstract specification is verified with the SPIN model checker in a usual way. The method has been successfully applied to verification of the MOSI protocol implemented in the Elbrus computer systems.

**Keywords**—multicore microprocessors, shared memory multiprocessors, cache coherence protocols, model checking, SPIN, PROMELA.

## I. INTRODUCTION

Shared memory multiprocessors (SMP) constitute one of the most common classes of high-performance computer systems. In particular, multicore microprocessors, which combine several processors (cores) on a chip, are widely used [1]. Nowadays, 8- and 16-core microprocessors are in mass production; hardware vendors have announced development of 48-, 80-, and even 100-core microprocessors. Multicore microprocessors and microprocessor systems are also designed by Russian companies such as MCST and INEUM, e.g., Elbrus-4C (4 cores, 2014) and Elbrus-8C (8 cores, 2015) [2].

The main problem arising in the development of SMP systems is ensuring *memory coherency*. As each processor contains a local cache, multiple copies of the same data may exist in the system: one copy in the main memory and several copies in the processors' caches. Modification of a copy should cause either the removal of the other copies or their coordinated modification. This is supported by the so-called *cache controllers*, i.e. memory devices connected into a network and cooperating in accordance with a special protocol, so-called *cache coherence protocol* [3].

Development of cache coherence mechanisms consists of two stages: design of a cache coherence protocol and its implementation in hardware. Considering high complexity of such mechanisms, the both stages are error-prone. To detect errors, methods for protocol verification and hardware verification are used [4]. Being especially critical, protocol bugs should be revealed before implementing the hardware. The widely recognized method for protocol verification is *model checking* [5]. It

is fully automated, but suffers from a principal drawback – it is not scalable due to the *state space explosion* problem. Verification of a cache coherence protocol for four and more processors is impossible (at least, highly problematic) with the traditional methods [6].

To overcome the problem and develop scalable verification technologies, researchers focus mostly on *parameterized model checking* [7]. The idea is to construct abstract models that are independent of the number of processors and may be verified with the existing tools. Correctness of the abstract model guarantees correctness of the original one (checking, however, may produce wrong error messages, so-called *false positives*). The proposed approach is also of that type. As a distinction, it supports the PROMELA language used in the SPIN model checker [8] and the message passing primitives. The method was successfully used for verifying the cache coherence protocols implemented in the Elbrus computer systems [2].

The paper is structured as follows. In Section 2, we analyze existing approaches to cache coherence protocol verification. In Section 3, we propose a method for constructing an abstract model of the protocol out of an original PROMELA model. In Section 4, we describe theoretical foundations of the suggested method. In Section 5, we provide a case study on using the method for verifying a MOSI protocol. In Section 6, we summarize our work and define further research directions.

## II. RELATED WORK

Classical model checking is inapplicable to cache coherence protocols with an arbitrary number of processors. There is an alternative approach, called *deductive verification*; however, it is hardly automated due to the need of so-called *inductive invariants* [9] and does not provide any diagnostic information if there are errors. Parameterized model checking seems to be a more promising approach. Two directions may be emphasized.

First, verification of a parameterized model (in essence, a family of models) can be reduced to the verification of a single model of the family. Corresponding methods are aimed at finding such number  $N$  that verification of the model for  $N$  components (processors, cache controllers, etc.) is sufficient for proving correctness in the general case. In [7], such kind of method is presented, and it is reported that  $N = 7$  is enough for the protocols having been examined. However, that value is too big to make the method applicable to cache coherence protocols of industrial SMP systems [6].

Second, a model (parameterized model) can be abstracted so as to reduce the state space size (make it independent of the number of components). Paper [10] introduces a method for abstracting a model from the exact number of *replicated identical components* (e.g., caches in which the cache line is in a given state). The technique significantly reduces the state space size; however, the use of a modified version of the Mur $\phi$  tool complicates its real-life application. A similar idea, called *(0,1, $\infty$ )-counter abstraction*, is employed in [11], [12], and [13]. Though the technique seems to be powerful, it often leads to overly detailed abstract models, which makes the approach inapplicable to complex protocols.

In [14], a general method for *compositional verification* is proposed. The idea is to replace a subset of identical components with an abstract one, so-called *environment*. Such replacement usually leads to false positives, and large amount of hand-work is required to eliminate them. In [15]–[18], the approach has been modified for cache coherence protocol verification. The suggested method is based on *syntactical transformations* of Mur $\phi$  models and *counterexample-guided abstraction refinement*. However, these works exhibit some drawbacks:

- 1) Mur $\phi$  does not support the message passing primitives, which complicates description of cache coherence protocols;
- 2) the class of protocols that can be verified with the method and the restrictions on Mur $\phi$  models are not clearly defined;
- 3) the tools are not in open access.

### III. THE SUGGESTED METHOD

The problem is as follows. Given a PROMELA model of a cache coherence protocol for some configuration of an SMP system (i.e. a model with a fixed number  $n > 2$  of processors), it is required to check the protocol correctness for an arbitrary configuration of the system (i.e. for any  $N \geq n$ ).

Models considered in this paper satisfy the following conditions (obtained from the verification practice and shown to be sufficient for specifying cache coherence protocols). The allowed statements are **if**, **do**, **goto**, = (*assignment*), ! (*send*), and ? (*receive*). Each guarded action is placed in an **atomic** block and therefore is executed atomically; **else** alternatives are absent. Assignments' right-hand sides contain only primary expressions, i.e. variables and constants; left-hand are variables and array elements (an array index is a primary expression). Atomic logic formulae are of the form  $x == c$  or  $B(ch)$ , where  $x$  is a variable (or array element),  $c$  is a constant,  $ch$  is a channel, and  $B$  is a predicate: **empty**, **full**, etc.

#### A. Generalization of the Original Model

From the conceptual point of view, a model consists of an unbounded number of replicated identical processes, so-called *basic processes*, and a fixed number of *auxiliary processes*. Without loss of generality we will assume that there is only one auxiliary process. All processes are enumerated from 0 to  $N$ , where  $N$  is the parameter: 0 is the identifier of the auxiliary process, while  $1, \dots, N$  are the identifiers of the basic processes. All

arrays used in the model (arrays of variables and arrays of channels) are of length  $N$  and indexed with the identifiers of the basic processes.

To generalize the original model to a parameterized one, the following rules are used:

- 1) each condition containing an array is either a conjunction or a disjunction of similar conditions on all elements of the array:

$\varphi\{i/1\} \wedge \dots \wedge \varphi\{i/n\}$  is interpreted as  $\forall i \in \{1, \dots, N\}: \varphi$ ;

$\varphi\{i/1\} \vee \dots \vee \varphi\{i/n\}$  is interpreted as  $\exists i \in \{1, \dots, N\}: \varphi$ ;

- 2) each sequence of statements  $\alpha\{i/1\}; \dots; \alpha\{i/n\}$  is interpreted as a loop **for** ( $i: 1 \dots N$ )  $\{\alpha\}$ .

Here,  $\varphi$  ( $\alpha$ ) is a formula (statement) containing an index  $i$  as a free variable, and  $\varphi\{i/t\}$  ( $\alpha\{i/t\}$ ) denotes the result of substitution of  $t$  for all occurrences of  $i$  in  $\varphi$  ( $\alpha$ ).

#### B. Clarification of Protocol Model

Let us consider a cache coherence protocol where request processing is coordinated by a *system commutator* of the *home processor* (the processor that owns the requested data). Accordingly, the PROMELA model contains two process types: *proc* is a cache controller (a basic process) and *home* is a home processor's commutator (an auxiliary process). Conventionally, the model deals with one cache line.

Broadly speaking, the cache coherence protocol is as follows. Each *proc* instance may initiate an operation on the cache line by sending a primary request to the *home* process. Upon its reception and analysis, *home* sends *snoop requests* to all processes except for the sender. After snoop reception, a *proc* sends a response to the sender (data or an acknowledgement that it has completed an action on the cache line). Having collected all of the answers, the sender informs *home* on the completion of the operation. As soon as the completion message is received, *home* can accept the next primary request.

It is worth emphasizing that at most one primary request is being processed at each moment of time. We will assume that values of global variables (e.g., a current sender identifier) are set by *home* upon reception of a primary request and do not change during its processing.

Each channel can be read by a single process; however, multiple processes are allowed to write into it. A channel is called *simple* if there is only one sender; otherwise, it is called *multiplexed*. Let  $C_{S \rightarrow r}$  be the set of channels with the reader  $r$  and senders from the set  $S$ . Channels are divided into three groups (singletons are written without brackets):

- 1)  $C_* = \bigcup_{j=0}^N C_{\{1, \dots, N\} \rightarrow j}$  is the set of multiplexed channels of capacity  $N$  used by *home* and *proc* to receive messages from the basic processes (e.g., a channel over which *home* receives primary requests, and channels over which processes receive responses);

- 2)  $C_{h \rightarrow p} = \bigcup_{j=1}^N C_{0 \rightarrow j}$  is the set of simple channels of positive capacity (that is defined by the protocol, but independent of  $N$ )

used by the basic processes to receive messages from *home* (e.g., channels over which *home* transmits snoop requests);

3)  $C_{p \rightarrow h} = \bigcup_{i=1}^N C_{i \rightarrow 0}$  is the set of simple channels of capacity 1 used by *home* to receive messages from the basic processes (e.g., channels over which senders inform *home* on operation completion).

Messages transmitted via channels are ordered pairs of the form  $(opc, i)$ , where *opc* is an operation code, and *i* is an identifier of the message sender.

A verified cache coherence property looks as follows:

$$\mathbf{G}\{\forall k, l \in \{1, \dots, N\}: (k \neq l) \rightarrow \varphi\{i/k, j/l\}\},$$

where **G** is an operator that requires its argument to be true in all reachable states of the model [5];  $\varphi$  is a formula with two free indices, *i* and *j*, that characterizes cache coherency in the corresponding caches. For MOSI protocols [3],  $\varphi$  is as follows:

$$\begin{aligned} & \neg(\text{cache}[i] = M \wedge \text{cache}[j] \neq I); \\ & \neg(\text{cache}[i] = O \wedge \text{cache}[j] = O); \end{aligned}$$

where *cache* is an array that stores the cache line states.

### C. Informal Description of the Method

The core of the proposed method is syntactical transformation of PROMELA code. The transformations change the process types; moreover, instead of  $N + 1$  processes, four processes remain: a modified *home* process (*home<sub>abs</sub>*), two modified *proc* processes (*proc<sub>abs</sub>*), and an environment process (*proc<sub>env</sub>*) that represents the remaining processes. Accordingly, the abstract model's initialization process is as follows (*ABS* is a constant distinct from 0, 1, and 2):

```
init {
  atomic {
    run homeabs(0);
    run procabs(1);
    run procabs(2);
    run procenv(ABS);
  }
}
```

The length of all arrays is changed from *N* to 2 (recall that arrays are indexed with the identifiers of the *proc* processes). Each array access is supplied with the guard  $i \leq 2$ , where *i* is the index of the element being accessed.

1) On read (in a condition), the atomic formula containing the array access, is replaced with *undef* (undefined value) if the index is rejected by the guard:

$$B(x[i], \dots) \Rightarrow (i \leq 2 \rightarrow B(x[i], \dots) : \text{undef}).$$

In PROMELA, the formula  $(B \rightarrow t_1 : t_2)$  corresponds to the conditional construct **if** *B* **then** *t<sub>1</sub>* **else** *t<sub>2</sub>* **fi**.

2) On write (in an assignment), the assignment to the array is placed inside the selection statement:

$$x[i] = t \Rightarrow \text{if} :: \text{atomic} \{i \leq 2 \rightarrow x[i] = t\} :: \text{else fi}$$

Assignments to global variables as well as conditions on global variables remain unchanged.

Channels of the set  $C_{h \rightarrow p}$  are represented as an array (let us denote it as *ch*). Similarly to other arrays, it is truncated to length two. Each atomic formula over *ch*[*i*], where  $i > 2$ , is replaced with *undef*, while each assignment to such a channel is removed. Channels of the sets  $C_*$  and  $C_{p \rightarrow h}$  are represented by individual channels instead of arrays.

Send statements are either unchanged or removed. A statement *ch!**m* in a process type *P* is removed only in the following cases:

- 1)  $ch \in C_{h \rightarrow e}$  and  $P = \text{home}_{abs}$ , where  $C_{h \rightarrow e} = \bigcup_{j=3}^N C_{0 \rightarrow j}$ ; e.g., *home<sub>abs</sub>* does not send snoop requests to *proc<sub>env</sub>*;
- 2)  $ch \in C_*$  и  $P = \text{proc}_{env}$ ; e.g., *proc<sub>env</sub>* does not send primary requests / snoop responses.

Receive statements may be left unchanged, modified, or removed. A statement *ch?**m* in a process type *P* is removed only in the following case:

$ch \in C_{h \rightarrow e}$  and  $P = \text{proc}_{env}$ ;  
e.g., *proc<sub>env</sub>* does not receive snoop requests.

Modification of *ch?**m* takes place solely in the following case:

$ch \in C_*$  and  $P \in \{\text{home}_{abs}, \text{proc}_{abs}\}$ .

The corresponding transformation replaces the guarded action **atomic** {*B*  $\rightarrow$  *ch?**m*} with the following selection statement:

```
if
  :: atomic {B'  $\rightarrow$  ch?m};
  :: atomic {m.opc = opc1; m.i = ABS}
  ...
  :: atomic {m.opc = opck; m.i = ABS}
fi
```

Here, *B'* is the result of *B* transformation, and *opc<sub>1</sub>*, ..., *opc<sub>k</sub>* are all possible operation codes that may be sent along *ch*.

Having performed the above transformations, logical formulae with *undef* (in essence, formulae of Kleene's strong three-valued logic) are transformed into classic logic formulae such that *undef* in the outer scope is interpreted as *true*. This is achieved by the obvious transformation *F*:

- 1)  $F(\varphi) = F'(\varphi, \text{true})$ ;
- 2)  $F'(\text{undef}, T) = T$ ;
- 3)  $F'(B, T) = B$ , where *B* is an atom distinct from *undef*;
- 4)  $F'(\neg\varphi, T) = \neg F'(\varphi, \neg T)$ ;
- 5)  $F'(\varphi \circ \psi, T) = F'(\varphi, T) \circ F'(\psi, T)$ , where  $\circ \in \{\wedge, \vee\}$ .

When transforming the PROMELA model the following optimizations are applied:

- 1) constant propagation and folding;
- 2) dead code elimination.

Here are simple examples of the optimizations:

- 1)  $(i \leq 2) \Rightarrow \text{true}$  in  $\text{home}_{abs}$  and  $\text{proc}_{abs}$ ;
- 2)  $(\text{true} \wedge B) \Rightarrow B$  and  $(\text{false} \wedge B) \Rightarrow \text{false}$ ;
- 3) **atomic**  $\{\text{true} \rightarrow \alpha\} \Rightarrow \alpha$  if  $\alpha$  cannot be blocked.

#### IV. THEORETICAL FOUNDATIONS

##### A. Basic Definitions

Let  $\text{Var}$  be a set of variables and  $\text{Chan}$  be a set of channels.  $\text{Data} = \text{Var} \cup \text{Chan}$  is referred to as the set of data. For each  $c \in \text{Chan}$ , a value  $|c| > 0$ , called *capacity*, is defined. A *data state*, or *state* for short, is a *valuation* of data, i.e. a mapping  $s$  that maps each variable  $v$  to the value  $s(v) \in \mathbb{N}$  and each channel  $c$  to the sequence of messages  $s(c) \in \mathbb{M}^*$  such that  $|s(c)| \leq |c|$ . The set of all states is denoted by  $S$ . A designated state  $s_0 \in S$  is called *initial*.

Let us assume that a language over data is formally defined. It includes logic formulae and statements, such as  $x = t$  (*assignment*),  $c ! m$  (*send*), and  $c ? m$  (*read*).

A *guard* is a formula; an *action* is a sequence of statements; a *guarded action* is a pair  $\gamma \rightarrow \alpha$ , where  $\gamma$  is a guard, and  $\alpha$  is an action. The guarded action  $\text{true} \rightarrow \epsilon$ , where  $\epsilon$  is the empty sequence of statements, is called *empty* and designated as  $\epsilon$ . The set of all guarded actions is denoted by  $\text{Act}$ . A guarded action  $\gamma \rightarrow \alpha$  is called *executable* in  $s \in S$  iff  $s \models \gamma$ .

A *process graph*, or *process* for short, is a triple  $\langle V, v_0, E \rangle$ , where  $V$  is a set of vertices,  $v_0 \in V$  is an *initial vertex*, and  $E \subseteq V \times \text{Act} \times V$  is a set of edges.

Process structure is defined by the control statements: **if** (*selection*), **do** (*repetition*), and **goto** (*jump*). Correspondence between code and process graphs is straightforward.

A *system* is a set of processes, i.e.  $\{\langle V_i, v_{0_i}, E_i \rangle\}_{i=0}^N$ . Herein after,  $P_i$  is a shortcut for  $\langle V_i, v_{0_i}, E_i \rangle$ . A *configuration* of  $\{P_i\}_{i=0}^N$  is a pair  $\langle l, s \rangle$ , where  $l: \{0, \dots, N\} \rightarrow \bigcup_{i=0}^N V_i$  such that  $l(i) \in V_i$  for all  $i \in \{0, \dots, N\}$  (so-called *control state*), and  $s \in S$ . The configuration  $\langle l_0, s_0 \rangle$ , where  $l_0(i) = v_{0_i}$  for all  $i \in \{0, \dots, N\}$ , is called *initial*.

The *state space* of  $\{P_i\}_{i=0}^N$  is a triple  $\langle C, c_0, T \rangle$ , where  $C$  is the set of all configurations of the system,  $c_0$  is the initial configuration, and  $T \subseteq C \times \left( \{0, \dots, N\} \times \left( \bigcup_{i=0}^N E_i \right) \right) \times C$  is a *transition relation* such that  $(\langle l, s \rangle, (i, (v, \gamma \rightarrow \alpha, v')), \langle l', s' \rangle) \in T$  iff:

- 1)  $l(i) = v$ ;
- 2)  $(v, \gamma \rightarrow \alpha, v') \in E_i$ ;
- 3)  $s \models \gamma$ ;
- 4)  $l' = (l \setminus \{i \mapsto v\}) \cup \{i \mapsto v'\}$ ;
- 5)  $s' = \llbracket \alpha \rrbracket(s)$ , where  $\llbracket \alpha \rrbracket: S \rightarrow S$  is the semantics of  $\alpha$ .

It is worth mentioning that the restrictions on the transition relation conform to the notion of *asynchronous parallelism*.

A configuration  $c$  is called *reachable* in  $\langle C, T, c_0 \rangle$  iff there is a path in  $T$  from  $c_0$  to  $c$ . A state  $s$  is called *reachable* iff a configuration  $\langle l, s \rangle$  is reachable for some  $l$ .

##### B. Abstraction of Processes and Systems

A *process transformation*, or *transformation* for short, is a function that maps one process to another.

Let  $\text{Data}_S = (\text{Var}_S \cup \text{Chan}_S) \subseteq \text{Data}$  be a set of *significant data* (variables and channels). States  $s$  and  $s'$  are called *equivalent* ( $s \sim s'$ ) iff  $s|_{\text{Data}_S} = s'|_{\text{Data}_S}$ .

A guarded action  $\gamma' \rightarrow \alpha'$  is referred to as an *abstraction* of  $\gamma \rightarrow \alpha$  in  $s \in S$  iff:

- 1) the truth of  $\gamma'$  is determined only by the significant data: for all  $s' \in S$  such that  $s' \sim s$ ,  $s' \models \gamma'$  iff  $s \models \gamma$ ;
- 2) the effect of  $\alpha'$  is determined only by the significant data: for all  $s' \in S$  such that  $s' \sim s$ , there holds  $\llbracket \alpha' \rrbracket(s') \sim \llbracket \alpha \rrbracket(s)$ ;
- 3) the guard  $\gamma'$  is weaker than  $\gamma$ :  $s \models \gamma \rightarrow \gamma'$ ;
- 4) the action  $\alpha'$  acts similar to  $\alpha$ :  $\llbracket \alpha' \rrbracket(s) \sim \llbracket \alpha \rrbracket(s)$ .

A set of guarded actions  $\{\gamma'_i \rightarrow \alpha'_i\}_{i=1}^m$  is referred to as an *abstraction* of  $\gamma \rightarrow \alpha$  in  $s \in S$  iff there exists  $i \in \{1, \dots, m\}$  such that  $\gamma'_i \rightarrow \alpha'_i$  is an abstraction of  $\gamma \rightarrow \alpha$  in  $s$ .

A guarded action  $\gamma' \rightarrow \alpha'$  (a set  $\{\gamma'_i \rightarrow \alpha'_i\}_{i=1}^m$ ) is referred to as an *abstraction* of  $\gamma \rightarrow \alpha$  iff  $\gamma' \rightarrow \alpha'$  ( $\{\gamma'_i \rightarrow \alpha'_i\}_{i=1}^m$ ) is an abstraction of  $\gamma \rightarrow \alpha$  in all states.

An *abstraction function* is a mapping  $f: \text{Act} \rightarrow 2^{\text{Act}}$  such that for all  $\gamma \rightarrow \alpha \in \text{Act}$ ,  $f(\gamma \rightarrow \alpha)$  is an abstraction of  $\gamma \rightarrow \alpha$ . The abstraction function  $I(\gamma \rightarrow \alpha) \equiv \{\gamma \rightarrow \alpha\}$  is called *trivial*.

It should be emphasized that abstraction takes into account context of a guarded action (the process edge, the process, and the model). It is assumed that each guarded action contains the context information.

Let  $P = \langle V, v_0, E \rangle$  be a process,  $f$  be an abstraction function,  $V'$  be some set, and  $R: V \rightarrow V'$  be a mapping. An *abstraction* of  $P$  induced by  $f$  and  $R$  is  $f(P, R) = \langle V', R(v_0), E' \rangle$ , where  $E'$  is defined as follows:

- 1) if  $(v, \gamma \rightarrow \alpha, u) \in E$  and  $f(\gamma \rightarrow \alpha) = \{\gamma'_i \rightarrow \alpha'_i\}_{i=1}^m$ , then  $\{(R(v), \gamma'_i \rightarrow \alpha'_i, R(u))\}_{i=1}^m \subseteq E'$ ;
- 2) no other edges belong to  $E'$ .

An abstraction  $f(P, R)$ , where  $R$  is a bijection is referred to as a *bijective abstraction*.

Besides transforming individual processes, transformations that merges several processes are of interest. Let us consider a particular kind of such transformations, where processes to be merged are identical.

Given a system  $\{P_i\}_{i=0}^N$ , the following denotations can be introduced ( $i \in \{0, \dots, N\}$ ):

- 1)  $\text{Use}_i$  is the set of variables read by  $P_i$ ;
- 2)  $\text{Use}_{L_i}$  is the set of variables read solely by  $P_i$ ;
- 3)  $\text{Def}_i$  is the set of variables assigned by  $P_i$ ;
- 4)  $\text{Def}_{L_i}$  is the set of variables assigned solely by  $P_i$ ;
- 5)  $\text{Var}_i = \text{Use}_i \cup \text{Def}_i$  is the set of variables of  $P_i$ ;
- 6)  $\text{Var}_{L_i} = \text{Use}_{L_i} \cup \text{Def}_{L_i}$  is the set of *local variables* of  $P_i$ ;
- 7)  $\text{Var}_G = \text{Var} \setminus (\bigcup_{i=0}^N \text{Var}_{L_i})$  is the set of *global variables*.



Similarly, the following sets of channels (including the *sets of local channels* and the *set of global channels*) can be defined:  $In_i, In_{L_i}, Out_i, Out_{L_i}, Chan_i$ , and  $Chan_{L_i}$ . In addition,

- 1)  $Data_i = Var_i \cup Chan_i$  is the set of data of  $P_i$ ;
- 2)  $Data_{L_i} = Var_{L_i} \cup Chan_{L_i}$  is the set of *local data* of  $P_i$ ;
- 3)  $Data_G = Var_G \cup Chan_G$  is the set of *global data*;

Processes are called *identical* if they can be transformed one another by renaming their local data.

Let  $\{P_i\}_{i=k_1}^{k_2}$  be a system of identical processes,  $g$  be an abstraction function,  $V'$  be some set, and  $R: V_{k_1} \rightarrow V'$  be a mapping. The process  $g(P_{k_1}, \dots, P_{k_2}; R) = g(P_{k_1}, R)$  is called a *unifying abstraction* of  $\{P_i\}_{i=k_1}^{k_2}$  induced by  $g$  and  $R$ .

Provided that the processes  $\{P_i\}_{i=k_1}^{k_2}$  operate simultaneously, some control states cannot be adequately represented by a single vertex of the abstraction  $g(P_{k_1}, \dots, P_{k_2}; R)$ . On the other hand, the serializability requirement (where at most one process is allowed to operate at each moment of time) is too strict. Let us assume that each process can be either *active* or *passive*. It is prohibited than two or more processes are active simultaneously. The passive mode is organized as follows: a request is received, the local data are updated, a response is sent, and the control is returned to the initial vertex.

Let  $V(E')$  be the set of all vertices of the edges from  $E'$ .

A process  $P = \langle V, v_0, E_A \cup E_P \rangle$  is referred to as a *bimodal process* with the set of *active edges*  $E_A$  and the set of *passive edges*  $E_P$  iff  $E_A \cap E_P = \emptyset$  and the graph  $\langle V(E_P), E_P \rangle$  is strongly connected.

Given a bimodal process  $P = \langle V, v_0, E_A \cup E_P \rangle$ , the following denotation can be introduced:  $V_A = V(E_A)$  and  $V_P = V(E_P)$  (generally speaking,  $V_A \cap V_P \neq \emptyset$ ).

The process  $g(P, R) = \langle V', v'_0, E' \rangle$ , where  $g$  is an abstraction function, and  $R: V \rightarrow V'$  is a mapping, is called a *serializing abstraction* of  $P$  iff  $R$  satisfies the following properties:

- 1)  $R(v) = v'_0$  for all  $v \in V_P \setminus V_A$ ;
- 2)  $R: V_A \rightarrow V'$  is a bijection;

and  $E'$  is defined as follows:

- 1) if  $(v, \gamma \rightarrow \alpha, u) \in E_A$  and  $g(\gamma \rightarrow \alpha) = \{\gamma'_i \rightarrow \alpha'_i\}_{i=1}^m$ , then  $\{(R(v), \gamma'_i \rightarrow \alpha'_i, R(u))\}_{i=1}^m \subseteq E'$ ;
- 2)  $(v'_0, \varepsilon, v'_0) \in E'$  (so-called  $\varepsilon$ -self loop);
- 3) no other edges belong to  $E'$ ;

and for every  $(v, \gamma \rightarrow \alpha, u) \in E_P$ , the empty guarded action  $\varepsilon$  is an abstraction of  $\gamma \rightarrow \alpha$  (i.e.  $\alpha$  depends on and affects solely insignificant data).

The nature of the serializing abstraction is removing all passive edges and replacing them with the  $\varepsilon$ -self loop  $(v'_0, \varepsilon, v'_0)$ . Being applied to identical bimodal processes, such abstraction makes them unimodal and serializable (at most one process is operating at each moment of time) and allows constructing an adequate unifying abstraction.

Let  $M = \{P_i\}_{i=0}^N$  be a system where all processes, except maybe  $P_0$ , are identical and bimodal;  $k \in \{0, \dots, N\}$  be some number;  $Data_S$  be significant data;  $V'_i$ , where  $i \in \{0, \dots, +1\}$ , be some sets;  $R_i: V_i \rightarrow V'_i$  be some mappings. Let  $f_i$ , where  $i \in \{0, \dots, k\}$ , and  $g$  be abstraction functions; at that,  $f_i(P_i, R_i)$  are bijective abstractions, while  $g(P_{k+1}, \dots, P_N; R_{k+1})$  is a serializing abstraction. Then, the system

$$M' = \{f_i(P_i, R_i)\}_{i=0}^k \cup \{g(P_{k+1}, \dots, P_N; R_{k+1})\}$$

is called an *abstraction* of  $M$ . Each process  $f_i(P_i; R_i)$ , where  $i \in \{0, \dots, k\}$ , is referred to as an *abstraction of the process*  $P_i$ , while the process  $g(P_{k+1}, \dots, P_N; R_{k+1})$  is referred to as an *abstraction of the environment*.

**Statement.** Let  $M = \{P_i\}_{i=0}^N$  and  $M' = \{P'_i\}_{i=0}^{k+1}$  be, respectively, a system and its abstraction. Given an arbitrary state  $s$ , if  $s$  is reachable in the state space of  $M$ , then there is a state  $s'$  reachable in the state space of  $M'$  and such that  $s' \sim s$ .

**Corollary.** Let  $M = \{P_i\}_{i=0}^N$  and  $M' = \{P'_i\}_{i=0}^{k+1}$  be, respectively, a system and its abstraction. Given an arbitrary formula  $\varphi$  over significant data, if  $\varphi$  is true (false) in all states reachable in the state space of  $M'$ , then  $\varphi$  is true (false) in all states reachable in the state space of  $M$ .

### C. Protocol Model Abstraction

This section defines abstraction functions used for protocol model transformation. The description is not quite formal: rigorous definition requires, first, formalization of PROMELA semantics and, second, involvement of formalisms for describing code transformations.

Let  $M = \{P_i\}_{i=0}^N$  and  $M' = \{P'_i\}_{i=0}^{k+1}$  be, respectively, a system (referred to as an *original protocol model*) and its abstraction (referred to as an *abstract protocol model*).

Let us recall that each message circulated in the model includes the sender's identifier. A state of a channel being written by a process  $P_i$ , where  $i \in \{k+1, \dots, N\}$ , as well as messages being read from the channel may contain identifiers from the set  $\{k+1, \dots, N\}$ . In the abstract model, there are no such identifiers: all elements  $\{k+1, \dots, N\}$  are mapped to  $ABS$  (usually,  $ABS = k+1$ ). Thus, the definition of the guarded action abstraction should be modified: states differing only as it is described above are considered to be *equivalent*.

Another issue is as follows. A state of a channel's buffer is not of importance until a message is read. The idea is to consider some channels (in particular, channels written by  $\{P_i\}_{i=k+1}^N$ ) as insignificant. In this case, a send statement can be replaced with  $\varepsilon$ . To preserve the abstraction properties, each read from the channel should be supplied (as alternative behavior) with the assignments of all possible values that could be sent by via the channel by the removed statement(s) to the message variable.

The suggested approach to protocol model abstraction implies the following restrictions on the input model:

- 1)  $Data_S = Data \setminus ((\bigcup_{i=k+1}^N Data_{L_i}) \cup (\bigcup_{i=k+1}^N Out_i))$ ;
- 2) for each  $i \in \{k+1, \dots, N\}$ , there holds  $Chan_i = Chan_{A_i} \cup Chan_{P_i}$ , where  $Chan_{A_i}$  and  $Chan_{P_i}$  are the sets of channels used, respectively, in the active and passive modes, and:

- a.  $\text{Chan}_{A_i} \cap \text{Chan}_{P_i} = \emptyset$ ;
- b.  $\text{Chan}_{A_i} \subseteq \text{Chan}_G$ ;
- c.  $\text{Chan}_{P_i} \subseteq \text{Chan}_{L_i}$ ;
- 3) only two channel predicates are in use: **empty** and **full**;
- 4) there are no control and data dependencies via variables between the processes  $\{P_i\}_{i=1}^N$ ;
- 5) each guarded action is closed under data dependencies via variables;
- 6) there are no data dependencies from the local data.

$M' = \{P'_i\}_{i=0}^{k+1} = \{f_i(P_i, R_i)\}_{i=0}^k \cup \{g(P_{k+1}, \dots, P_N; R_{k+1})\}$ , the abstract model, is constructed as follows (the description below can be viewed as a definition of the mappings  $R_i$  and the abstraction functions  $f_i$  and  $g$ ). Initially, each process  $P'_i$ , where  $i \in \{0, \dots, k+1\}$ , is isomorphic to  $P_i$ :  $P'_i = I(P_i, R_{0_i})$ , where  $I$  is the trivial abstraction function, and  $R_{0_i}: V_i \rightarrow V'_i$  is a bijection. Then, the following transformations are applied to  $P'_{ABS} = P'_{k+1}$  and the rest processes:

- 1) all passive edges of  $P'_{ABS}$  are removed and replaced with the  $\varepsilon$ -self loop;
- 2) when removing a passive edge whose action contains a read from some channel  $c$  (a write to some channel  $c$ ):
  - a. in  $\{P'_i\}_{i=0}^k$ , for all  $j \in \{k+1, \dots, N\}$ , all writes to  $c_j$  (all reads from  $c_j$ ), where  $c_j$  is a channel of  $P_j$  corresponding to  $c$  (the processes are identical), are removed;
  - b. when removing a read to a message  $m$ :
    - i. in the guards dependent on  $m$ , the minimal subformulae dependent on  $m$  are replaced with *undef*;
- 3) the active edges of  $P'_{ABS}$  are processed as follows:
  - a. all assignments to the local variables are removed;
  - b. when removing an assignment to a local variable  $x$ :
    - i. in the guards dependent on  $x$ , the minimal subformulae dependent on  $x$  are replaced with *undef*;
  - c. each read from a global channel  $c$  is not modified:
    - i. in  $\{P'_i\}_{i=0}^k$ , writes to  $c$  are not modified;
  - d. each write to a global channel  $c$  is removed:
    - i. in  $\{P'_i\}_{i=0}^k$ , each read  $c ? m$  is supplemented with the alternatives  $\{m = v_j\}_{j=1}^t$ , where  $\{v_j\}_{j=1}^t$  contains all possible values that  $P'_{ABS}$  can send via  $c$ .

**Statement.** The processes  $\{f_i(P_i, R_i)\}_{i=0}^k$  (constructed as it is described above) are bijective abstractions, while the process  $g(P_{k+1}, \dots, P_N; R_{k+1})$  is a serializing abstraction.

## V. CASE STUDY

The proposed method was used to verify the MOSI family cache coherence protocols implemented in the Elbrus computer systems. The developed PROMELA model supports memory accesses the types Write Back, Write Through, and Write Combined. The experiments were performed on an Intel Core i7-4771 machine with a clock rate of 3.5 GHz. The verified properties are as follows:

- 1)  $\mathbf{G}\{\neg(\text{cache}[1] = M \wedge \text{cache}[2] = M)\}$ ;
- 2)  $\mathbf{G}\{\neg(\text{cache}[1] = O \wedge \text{cache}[2] = O)\}$ ;
- 3)  $\mathbf{G}\{\neg(\text{cache}[1] = M \wedge \text{cache}[2] \in \{O, S\})\}$ .

Table 1 and Table 2 show resources consumed for checking the property (1), respectively, on the original and the abstract model for  $n = 3$ . Note that in this case the proposed abstraction preserves the number of processes: *home*(0), *proc*(1), and *proc*(2) are replaced with their abstract counterparts, while *proc*(3) is replaced with *proc<sub>env</sub>*(ABS).

Table 1. Required resources — original model

SPIN optimization	State space size	Memory consumption	Verification time
<i>Absent</i>	$5.1 \times 10^6$	682 Mb	9 s
<i>COLLAPSE</i>	$5.1 \times 10^6$	328 Mb	15 s

Table 2. Required resources — abstract model

SPIN optimization	State space size	Memory consumption	Verification time
<i>Absent</i>	$2.2 \times 10^6$	256 Mb	3.7 s
<i>COLLAPSE</i>	$2.2 \times 10^6$	108 Mb	6.2 s

The tables show that even for  $n = 3$  there is a gain in state space size and memory consumption. Meanwhile, verification of the constructed abstract model means verification of the protocol for any  $n \geq 3$ . The task has been reduced to checking of  $\sim 10^6$  states, which consumes  $\sim 100$  Mb of memory.

## VI. CONCLUSION

Many high-performance computers and most multicore microprocessors use shared memory and utilize complicated caching mechanisms. To ensure that multiple copies of data are kept up-to-date, cache coherence protocols are employed. Errors in the protocols and their implementations may cause serious consequences such as data corruption and system hanging. This explains the urgency of the corresponding verification methods.

The main problem when verifying cache coherence protocols is state explosion. The article proposes an approach to overcome the problem and make verification scalable. A protocol model is described in PROMELA, a widely spread language in the verification community. The method is aimed at transforming the model so as the result is independent of the number of processors and can be verified by the SPIN model checker.

The approach was successfully applied to the verification of the MOSI family cache coherence protocols implemented in the Elbrus computer systems.

Directions of future work include extension of the method with counterexample-guided abstraction refinement (CEGAR), development of an open-source tool for syntactical transformations of PROMELA models (an experimental prototype is already available), and creation of a model-based technology for verifying cache coherence protocols and memory management units of industrial computer systems.

# REFERENCES

- [1] Patterson D.A., Hennessy J.L. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 2013. 800 p.
- [2] Kim A.K., Perekatov V.I., Ermakov S.G. Mikroprocessory i vychislitel'nye komplekсы semeystva «El'brus» (Microprocessors and computer systems of the Elbrus family). SPb.: Piter, 2013. 272 p.
- [3] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011. 195 p.
- [4] Kamkin A.S., Petrochenkov M.V. Sistema podderzhki verifikatsii realizatsij protokolov kogerentnosti s ispol'zovaniem formal'nyh metodov (A system to support formal methods-based verification of coherence protocol implementations) // Voprosy radioelektroniki. Ser. EVT. 2014. Vyp. 3. P. 27-38.
- [5] Clarke E.M., Grumberg O., Peled D.A. Model Checking. MIT Press, 1999. 314 p.
- [6] Burenkov V.S. Analiz primenimosti instrumenta SPIN k verifikatsii protokolov kogerentnosti pamyati (An analysis of the SPIN model checker applicability to cache coherence protocols verification) // Voprosy radioelektroniki. Ser. EVT. 2014. Vyp. 3. P. 126-134.
- [7] Emerson E.A., Kahlon V. Exact and Efficient Verification of Parameterized Cache Coherence Protocols // Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, 2003. P. 247-262.
- [8] Holzmann, G.J. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional. 2003. 608 p.
- [9] Park S., Dill D.L. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions // Annual ACM Symposium on Parallel Algorithms and Architectures, 1996. P. 288-296.
- [10] Ip C.N., Dill D.L. Verifying Systems with Replicated Components in Murphi // International Conference on Computer Aided Verification, 1996. P. 147-158.
- [11] Pnueli A., Xu J., Zuck L. Liveness with  $(0, 1, \infty)$ -Counter Abstraction // International Conference on Computer Aided Verification, 2002. P. 107-122.
- [12] Clarke E., Talupur M., Veith H. Environment Abstraction for Parameterized Verification // Verification, Model Checking, and Abstract Interpretation, 2006. LNCS, Vol. 3855. P. 126-141.
- [13] Clarke E., Talupur M., Veith H. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems // International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2008. P. 33-47.
- [14] McMillan K. Parameterized Verification of the FLASH Cache Coherence Protocol by Compositional Model Checking // Conference on Correct Hardware Design and Verification Methods, 2001. P. 179-195.
- [15] Chou C.-T., Mannava P.K., Park S. A Simple Method for Parameterized Verification of Cache Coherence Protocols // Formal Methods in Computer-Aided Design, 2004. LNCS, Vol. 3312, P. 382-398.
- [16] Krstic S. Parameterized System Verification with Guard Strengthening and Parameter Abstraction // International Workshop on Automated Verification of Infinite-State Systems, 2005.
- [17] Talupur M., Tuttle M.R. Going with the Flow: Parameterized Verification Using Message Flows // Formal Methods in Computer-Aided Design, 2008. P. 1-8.
- [18] O'Leary J., Talupur M., Tuttle M.R. Protocol Verification Using Flows: An Industrial Experience // Formal Methods in Computer-Aided Design, 2009. P. 172-179.

# A Model Checking-Based Method of Functional Test Generation for HDL Descriptions

Mikhail Lebedev, Sergey Smolov

Institute for System Programming of the Russian Academy of Sciences (ISP RAS)

Moscow, Russian Federation

Email: {lebedev, smolov}@ispras.ru

**Abstract**—Automated test generation is a promising direction in hardware verification research area. Functional test generation methods based on models are widespread at the moment. In this paper, a new functional test generation method is proposed and compared to existing solutions. It is based on automated extraction of high-level decision diagram models from hardware design's source code. Extracted models are then automatically translated into an input format of the nuXmv model checking tool and checked by it. The aim of model checking is to produce tests that cover all the transitions of an extended finite state machine model that is also automatically extracted from the source code. Functional tests are extracted from the model checker execution results. Experiments show advantages of the proposed method.

**Keywords**—hardware design; functional verification; static analysis; test generation; guarded action; high-level decision diagram; extended finite state machine; model checking.

## I. INTRODUCTION

Functional verification is an expensive and time-consuming stage of hardware design process [1]. Due to hardware designs increasing complexity, automated test generation seems to be important and challenging. To avoid design complexity, automated verification methods often utilize mathematical abstractions of system properties and behavior, so-called *models*. Models can be created manually or automatically extracted from the system's source code. Automated verification methods based on model extraction from the HDL (Hardware Description Language – a collective name for several languages described below) source code are considered in this paper. Models can be based on the following formal descriptions: finite-state machines, decision diagrams, Petri nets [2], etc.

Model checking [3] is an approach to set up the correspondence between the model of the system and formal conditions (specifications). For every specification a model checker tries to produce a *counterexample* – an input stimuli sequence that leads the system into a specification-contradicting state. Counterexample construction is often used for functional test generation purposes.

Proof of equivalence of a model and the corresponding system is an important issue when model checking is used for hardware verification. There is no need in such proof when the

model is automatically extracted from the system's source code and translated into the model checker's format.

A method of functional test generation for hardware is proposed in this paper. It is based on automatic extraction of High-Level Decision Diagrams (HLDD) [4] from the system's source code. Synthesizable sets of VHDL [5] and Verilog [6] HDLs are supported. Extracted models are then automatically translated into SMV (Symbolic Model Verifier) language supported by the nuXmv [7] model checker. Extended Finite State Machine (EFSM) transition constraints are used as specifications for model checking. EFSM model is also extracted from the system's source code. Counterexamples built by the nuXmv model checker are then translated into an HDL testbench which can be simulated by an HDL simulator.

The rest of the paper is organized as follows. Section II contains a review of functional test generation methods based on model extraction from the HDL source code. In Section III basic definitions are given. Section IV contains HLDD construction and test generation methods. Section V is dedicated to the experimental results. Section VI concludes the paper.

## II. RELATED WORKS

The idea of model extraction from the HDL source code and following test generation is not new. A prototype of CV tool for VHDL description model checking is presented in [8]. The tool's execution process consists of five stages. On the first stage a VHDL description is parsed and an internal representation is constructed. A Binary Decision Diagram (BDD) based model is built on the second stage. On the third stage a CTL-based specification is parsed. The specification language syntax is described in the paper. On the fourth stage the specification parsing result and the BDD-based model are passed to the CBMC [9] model checker. On the final stage the model checker output is translated into tests which can be executed by the HDL simulator. It is stated that BDD-based model size plays the key role in the model checking process. Model size reduction heuristics usage is suggested to avoid state space explosion but no heuristics are offered in the paper.

In [10], extraction of the EFSM model and generation of tests that cover all the model transitions are described (so-called FATE method). This method assumes that the user provides additional information for the tool about signal

semantics (for example, which of the signals encodes state). The EFSM extraction process contains several stages of transition structure simplification (embedded conditions elimination, compatible transitions union, dataflow dependency analysis). The test generation method is based on the state graph traversal through random walk and *backjumping* techniques.

In [11] an improved modification of method [10] is proposed. Optimizations described concern path reachability (*weakest precondition* [12] is used instead of the approximate approach) and test filtering tasks. A new functional test generation method called RETGA is also proposed in [11]. This method is based on the algorithm [13] for automated EFSM model extraction from HDL descriptions. The algorithm does not require additional information about signals/variables semantics; for state and clock-like variable detection it uses heuristics based on dataflow dependencies. Experiments have shown that RETGA method produces shorter tests with higher HDL code coverage than FATE and even improved FATE do.

It should be noted, that state graph traversal techniques (that FATE and RETGA methods use) do not guarantee coverage of all the EFSM model transitions. One of the problems concerns *counter* [11] variables that are defined in transition loops and used in transition guards, so an EFSM simulation engine needs to recognize at which value of the counter it is going to finish the loop execution.

### III. BASIC DEFINITIONS

Suppose that all models described below run in discrete time that implies clock presence. Clock  $C$  is a set of events  $\{c_1, \dots, c_k\}$  where an event  $c = \{\text{signal}, \text{edge}\}$  is a pair, consisting of a one-bit *signal* (so-called *clock pulse*) and a type of registration called *edge* (i.e. *positive edge* when *signal* changes its value from 0 to 1 and *negative edge* otherwise).

Let  $V$  be a set of *variables*. A *valuation* is a function that associates a variable  $v \in V$  with a value  $[v]$  from the corresponding domain. Let  $Dom_V$  be a set of all valuations of  $V$ . A *guard* is a Boolean function defined on valuations ( $Dom_V \rightarrow \{\text{true}, \text{false}\}$ ). An *action* is a transform of valuations ( $Dom_V \rightarrow Dom_V$ ). A pair  $\gamma \rightarrow \delta$ , where  $\gamma$  is a guard and  $\delta$  is an action, is called a *guarded action* (GA). It is implied that there is a description of every function in some HDL-like language (thus, we can reason about not only semantics, but syntax).

Let guarded actions be *synchronized* [14], if each GA is associated with a clock. A system  $\{\langle C^{(i)}, \gamma^{(i)} \rightarrow \delta^{(i)} \rangle\}_{i=1,l}$  of synchronized guarded actions can be represented by an oriented acyclic graph  $G = (N, E, C)$  called *Guarded Actions Decision Diagram* (GADD). Here  $N$  is a set of graph nodes,  $E$  is a set of graph edges, and  $C$  is a clock.  $N$  contains two non-intersecting subsets: a set  $N_s$  of non-terminal nodes  $n_s$  that are marked by expressions  $\gamma(n_s)$ ; a set  $N_t$  of terminal nodes  $n_t$  that are marked by actions  $\delta(n_t)$ . Graph edges can start from non-terminal nodes only and finish either in terminal or in non-terminal nodes. Edges  $e \in E$  are marked by sets  $Val(e, n)$  of accepted values  $\gamma(n)$  (here edge  $e$  is an outgoing edge for the node  $n$ ,  $e \in Out(n)$ ). The node  $n \in N_s$  can have no more than

one  $e_d \in Out(n)$  that is marked by *default* keyword – it means that for this path in  $G$  an expression  $\gamma(n)$  equals to a value that does not belong to any marking sets of the other edges outgoing from the node  $n$ . Supposing that the GADD contains exactly one root node  $n_{root}$  (the node without incoming edges,  $In(n_{root}) = \emptyset$ ), a set of *paths* from the root node to all the terminal nodes produces a system of synchronized guarded actions. For example, the  $i^{th}$  path, including  $n_1^{(i)}, \dots, n_m^{(i)}$  nodes and  $e_1^{(i)}, \dots, e_{m-1}^{(i)}$  edges ( $n_1^{(i)} \equiv n_{root}$ ,  $n_m^{(i)} \in N_t$ ,  $e_k^{(i)} \in Out(n_u^{(i)}) \cap In(n_{u+1}^{(i)})$ ,  $u = 1, \dots, m-1$ ), defines a guarded action with  $p_1^{(i)} \dots p_{m-1}^{(i)}$  ( $p_r^{(i)} = \text{AND}(\gamma(n_r) == q)$ ,  $r = 1, \dots, m-1$ ,  $q \in Val(e_r, n_r)$ ) conjunction as a guard and  $\delta(n_m^{(i)})$  as an action. The guarded action clock is a subset of the GADD clock.

In [4] a definition of a high-level decision diagram (HLDD) is given and is shown that every variable of an HDL description can be represented by a function  $v = f(v_1, \dots, v_n) = f(V)$  in terms of HLDD  $H_v$ . Let  $Z(v)$  be a finite set of all possible values of a variable  $v \in V$ . A *High-Level Decision Diagram* for  $v$  is an oriented acyclic graph  $H_v = (M, \Gamma, V)$  where  $M$  is a set of nodes, and  $\Gamma$  is a mapping  $M \rightarrow 2^M$ . Let  $\Gamma(m)$  be a set of *subsequent* nodes of the node  $m \in M$  and  $\Gamma^{-1}(m)$  be a set of *preceding* nodes of the node  $m$ . A node  $m_0$  of the graph  $H_v$  is said to be *initial* if the set of its preceding nodes is empty:  $\Gamma^{-1}(m_0) = \emptyset$ .  $M$  consists of two non-intersecting subsets:  $M_n$  for non-terminal nodes and  $M_t$  for terminal nodes. All the non-terminal nodes  $m_c \in M_n$  are marked by variables  $v(m_c) \in V$  and meet the following condition:  $2 \leq |\Gamma(m_c)| \leq |Z(v(m_c))|$ . This means that  $m_c$  has at least two subsequent nodes but not more than the number of possible values of  $v(m_c)$ . All the terminal nodes  $m_k \in M_t$  are marked by functions  $v(m_k) = f_k(V_k)$ ,  $f_k(V_k) \in F$  ( $V_k \subseteq V$ ). Usually some of these functions are trivial and equal either to variables  $v_k \in V$  or to constants. All the edges are marked by sets of accepted values of variables in the same manner as in the GADD definition; the semantics of the default edges is also similar.

On every tick of the clock, the HLDD  $H_v$  assigns a value to the target variable  $v$  through an *activation* procedure. Starting from the initial node  $m_0$  it calculates values of the variables which mark non-terminal nodes. For a value  $e$  of the variable  $v(m_c)$ ,  $e \in Z(v(m_c))$ , the corresponding edge from the node  $m_c \in M$  to the subsequent node  $m^e \in \Gamma(m_c)$  is activated. A vector  $V^t$  of variable values activates the path  $l(m_0, m_k)$  from  $m_0$  to the terminal node  $m_k$  marked by the function  $f_k(V_k)$  that determines the value of the target variable  $v$ .

### IV. HLDD MODEL CONSTRUCTION AND TEST GENERATION METHOD

The proposed test generation method consists of the following steps:

1. HDL (VHDL/Verilog) description parsing and GADD model construction.
2. HLDD model construction using the GADD model.

3. HLDD model and specification translation into the nuXmv model checker input language (SMV model) [16].
4. SMV model checking by the nuXmv model checker and translation of counterexamples into HDL tests.

The first step has been implemented in [13] so we will start from the second step. Note that all the actions which mark the terminal nodes of the GADD model are represented in the *static single assignment* (SSA) [15] form.

#### A. HLDD model construction

GADD and HLDD models preserve the module structure of the original HDL description. Every HDL description process is represented by a single GADD. The GADD  $G = (N, E, C)$  is used as a basis for HLDD construction for every non-input variable of the process. HLDD construction algorithm pseudo code is listed below:

```

proto = new;
for node ∈ N do
  hldd_node = transform_node(node);
  proto.add(hldd_node);
end
copy_edges(E, proto);
for (v : non_input_variables(G)) do
  hldd = proto.keep_assigns(v);
  hldd.add_missing_terminals();
  hldd.transform_identical_assigns();
end

```

At the first step the HLDD prototype *proto* is created. GADD nodes are transformed into HLDD nodes with the help of the *transform\_node* method and added to the prototype. Terminal GADD nodes  $n_i \in N_t$  are transformed into terminal HLDD nodes  $m_k \in M_t$ . Every terminal node  $n_i$  marked by multiple assignment action  $\delta(n_i)$  is transformed into a sequence of nodes. Every node in this sequence is marked by a corresponding single assignment  $a_k$ . Every terminal HLDD node is marked by a target variable  $v_k$  (which is the left-hand side of  $a_k$ ) and a function  $f_k(V_k)$  (which is the right-hand side of  $a_k$ ).

Non-terminal GADD nodes  $n_s \in N_s$  are transformed into non-terminal HLDD nodes  $m_c \in M_n$ . Guard  $\gamma$  which marks the node  $n_s$  is replaced by a new variable *guard*( $m_c$ ) which marks the node  $m_c$ . The new HLDD that contains a single terminal node marked by  $\gamma$  is created for this variable (*create\_variable\_from\_switch* method). GADD edges are transformed into HLDD edges by the *copy\_edges* method. The corresponding values are not changed.

Then for every non-input variable  $v$  the HLDD *hldd* is created which is actually a modified copy of *proto*. The *keep\_assigns* method removes from  $M_t$  the terminal nodes which are not marked by  $v$ . After that the *add\_missing\_terminals* method adds new terminal nodes marked by  $f(v) = v$  to the edges which lack the subsequent terminal nodes. This means that the value of  $v$  does not change if any path to such node is activated. The *transform\_identical\_assigns* method searches for such non-terminal nodes  $m_c$  whose reachable terminal nodes are marked

by the same function  $f_k(v_k)$ , and replaces  $m_c$  and its reachable subgraph with the only terminal node marked by  $f_k(v_k)$ .

Consider an example of the HLDD model construction for a simple VHDL description. This description contains a single module and a single process. The module interface consists of input variables *clk*, *rst*, *x*, *y* and an output variable *res* (all of 1-bit size). The process contains two internal variables: a 1-bit size vector *cnt* and an integer *state* (that can be assigned either 0 or 1). The source code of the process is listed below:

```

process (clk, rst, x, y)
  variable cnt: std_logic;
  variable state: integer range 0 to 1;
begin
  if (rst = '1') then
    cnt := '0';
    state := 0;
  elsif (clk = '1') then
    if (state = 1) then
      cnt := x or y;
      state := 0;
    elsif (state = 0) then
      cnt := x and y;
      state := 1;
    end if;
    res <= cnt;
  end if;
end process;

```

Fig. 1 shows the GADD model of the process. Non-terminal nodes of the GADD are shown as diamonds and correspond to branch expressions. Terminal nodes are shown as rectangles and correspond to basic blocks. Outgoing edges from the non-terminal nodes are marked by possible values of the branch expressions. Note that the *default* edge on Fig. 1 is unreachable because the *state* variable can only take the value of 0 or 1. The clock of the GADD is formed by events of *clk*, *rst*, *x* and *y* signals.

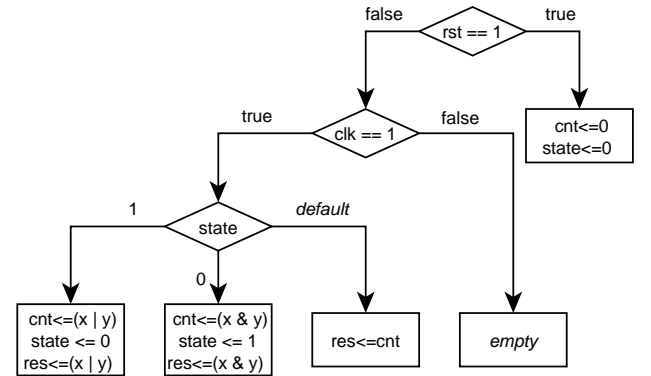


Fig. 1. GADD model.

Fig. 2 shows the HLDD prototype. Expressions which mark the non-terminal nodes are replaced by *guard0*, *guard1*, *guard2* variables.



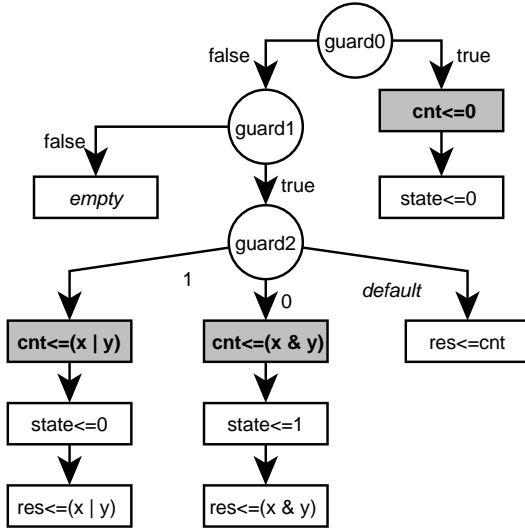


Fig. 2. HLDD prototype.

Consider the HLDD construction for the *cnt* variable. Terminal nodes marked by *cnt* are highlighted in grey on Fig. 2. Terminal nodes which are not marked by this variable are removed. New terminal nodes marked by *cnt* are added to the free non-terminal node edges (this means that the value of *cnt* does not change on these paths). The final HLDD is represented on Fig. 3. Similar diagrams are constructed for the other non-input variables of the HDL description (in our example those are: *state* and *res*).

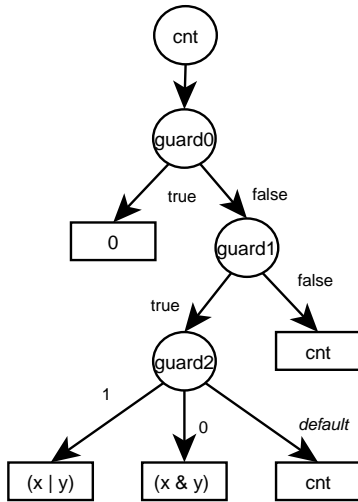


Fig. 3. HLDD of a *cnt* variable.

### B. SMV model construction and checking

The constructed HLDD model is translated into an SMV language description. Hardware design module structure is preserved. Any variable constraints (like the range of possible values that is specified for the *state* variable) and their initial

values described in the HDL description are added to the SMV model.

Specification construction is based on the EFSM model extracted from the same HDL description. Formal definition of the EFSM model and its extraction algorithm from an HDL description are presented in [13]. Here we provide only the informal definition. Extended finite-state machine (EFSM). It contains sets of inputs, outputs and internal variables. EFSM transitions are marked by guard expressions which depend on input and internal variable values and by actions which can change internal and output variable values. A transition can be enabled only if its guard becomes *true*. When a transition is enabled, its action is executed. Specifications used by the proposed method are represented as negations of the EFSM transition guards. Negation is used to make the model checker build a counterexample – a sequence of data states and input stimuli which contradicts the specification (and thus satisfies the corresponding guard).

The SMV model along with the specifications is checked by the nuXmv model checker. Output counterexamples are translated into a test set aimed at covering reachable EFSM transitions.

Below you can see the HLDD-to-SMV translation result for the *cnt* and *guard0* variables. NuXmv-compatible SMV language format is used. The description consists of the variable declaration section (VAR) and the assignment section (ASSIGN). The *init* construct defines the initial value of a variable. The *next* construct defines the value of a variable in the next model state. The assignment (“:=”) defines the value of a variable in the current model state. Numeric values in the example are of bit vector type and are represented by “0<type><size>\_<value>” construct.

```

VAR
  cnt : word[1];
  guard0 : boolean;
  ...
ASSIGN
  init(cnt) := 0d1_0;
  ...
ASSIGN
  next(cnt) :=
    case
      (guard0 = TRUE) : 0d1_0;
      (guard0 = FALSE) :
        case
          (guard1 = TRUE) :
            case
              (guard2 = 0sd32_1) : (x | y);
              (guard2 = 0sd32_0) : (x & y);
              TRUE : cnt;
            esac;
          (guard1 = FALSE) : cnt;
        esac;
      ...
  guard0 := (rst = 0d1_1);
  guard1 := (clk = 0d1_1);
  guard2 := state

```

The example of an SMV specification is listed next:

```
LTLSPEC ! F ((state = 0sd32_0) & (clk = 0d1_1)
& !(rst = 0d1_1));
```

EFSM transition reachability condition consists of the *state* variable constraint (which determines the source state of the transition) and the guard condition depending on the *clk* and *rst* variables.

The nuXmv model checker generates the following counterexample for this specification:

**Trace Type: Counterexample**

-> **State: 1.1** <-

```
SAMPLE.process.state = 0sd32_0
SAMPLE.process.cnt = 0ud1_0
SAMPLE.process.guard2 = 0sd32_0
SAMPLE.process.guard1 = FALSE
SAMPLE.process.guard0 = FALSE
SAMPLE.process.res = 0ud1_0
clk = 0ud1_0
y = 0ud1_0
x = 0ud1_0
rst = 0ud1_0
```

-> **State: 1.2** <-

```
SAMPLE.process.guard1 = TRUE
clk = 0ud1_1
```

The first state shows the initial values assigned to the variables. The second state shows only the values that have changed. We can see that the second state contradicts the given SMV specification: *clk* is equal to 1, while the *rst* and *state* variables are equal to 0.

## V. EXPERIMENTAL RESULTS

The proposed test generation method was implemented as a part of the HDL Retrascope 0.2.1 software tool [17]. Java language was used for development along with the Fortress formulae manipulation library [18]. Some HDL descriptions from the ITC'99 benchmark [19] were used for testing of the proposed approach.

The nuXmv model checker supports both symbolic model checking and bounded model checking [21] methods. In some cases symbolic model checking needed too much time and computer resources because of the state explosion (for example, B04, B10 and B11 designs). Bounded model checking could manage this problem by exploring the model state space only up to some bound. However, bound value affects the model checking results (not all the counterexamples may be obtained at the specified bound). So in some cases the bound size was iteratively increased in order to get all possible counterexamples.

Generated tests were simulated by the QuestaSim HDL simulator [20]. Test properties (length and source code coverage) were compared to existing test generation methods like FATE [10], RETGA [11] (these methods are based on EFSM model extraction from the HDL descriptions and are targeted at covering the EFSM model transitions) and random test generation.

TABLE I. contains information about the ITC'99 designs that were used for test generation: their source code size and the corresponding SMV model size (without specifications). Size is given in lines of code.

TABLE I. HDL DESCRIPTION AND SMV MODEL SIZE

Design	HDL	SMV
B01	102	207
B02	70	143
B03	134	637
B04	101	809
B06	127	442
B07	92	370
B08	88	315
B09	100	263
B10	167	755
B11	118	368

TABLE II. contains the test length information. Test length is given in clock cycles. The length of tests generated by the random generation method corresponds to the point when the test coverage growth stops (maximum length was chosen as 1000000 clock cycles). The sign “-” means that the corresponding method failed to generate tests for the corresponding HDL design.

TABLE II. TEST LENGTH

Design	FATE	RETGA	SMV	Random
B01	115	49	69	300
B02	62	33	47	80
B03	-	-	504	2000
B04	104	36	67	200
B06	198	76	88	700
B07	246	166	249	1000
B08	31	52	31	1000000
B09	19	231	84	1000000
B10	173	135	134	650000
B11	101	721	194	1000000

In 5 of 10 cases tests generated by the proposed method are shorter than tests generated by the FATE method and longer than RETGA tests. The rest tests are either of comparable length with the leader (RETGA), or tests generated by the FATE method provide lower coverage. Definitive conclusion about the advantages or disadvantages of the proposed method in comparison with the RETGA method cannot be made using the selected HDL description set.

Notice that unlike the FATE and RETGA methods the proposed method isn't based on EFSM traversal. So it was able to generate the test for B03 design in contrast to those methods (EFSM extracted from this design is too complex for traversal).

TABLE III. shows the HDL source code statement coverage in comparison to the FATE, RETGA and random generation methods.

TABLE III. SOURCE CODE STATEMENT COVERAGE

Design	FATE	RETGA	SMV	Random
B01	97,14%	100%	100%	100%
B02	100%	100%	100%	100%
B03	-	-	100%	100%
B04	100%	100%	100%	100%
B06	100%	100%	100%	100%
B07	93,93%	93,93%	93,93%	84,85%
B08	81,81%	100%	100%	90,91%
B09	35,29%	100%	100%	61,77%
B10	95,94%	100%	100%	97,29%
B11	69,23%	94,87%	94,87%	87,18%

TABLE IV. shows the HDL source code branch coverage in comparison to the FATE, RETGA and random generation methods.

TABLE IV. SOURCE CODE BRANCH COVERAGE

Design	FATE	RETGA	SMV	Random
B01	96,15%	100%	100%	100%
B02	100%	100%	100%	100%
B03	-	-	100%	100%
B04	100%	100%	100%	100%
B06	100%	100%	100%	100%
B07	94,73%	94,73%	94,73%	73,69%
B08	76,92%	100%	100%	84,62%
B09	35,71%	100%	100%	57,15%
B10	90,47%	100%	100%	97,61%
B11	71,87%	96,87%	96,87%	90,63%

The proposed method achieved the same code coverage as the RETGA method at the specified set of HDL descriptions. B07 and B11 HDL description coverage is less than 100% because of the unreachable code in these designs.

## VI. CONCLUSION AND FUTURE WORK

The functional test generation method based on automated HLDD model extraction and checking with nuXmv is presented in this paper. The main advantage of this method is its flexibility in choosing a test target (through using different kinds of specifications). EFSM transition coverage is presented for comparison to the other test generation methods (FATE, RETGA). Any other specifications can be formulated and checked in order to get a test aimed at covering the corresponding property of a model.

The presented implementation of the proposed approach does not produce shorter tests than existing approaches on the chosen hardware design set. Simple optimizations (like test filtering) can be helpful and are going to be implemented in the nearest future.

Future work is focused on applying the method to more complex hardware designs (including Verilog-based). In this case complexity is defined by the number of execution paths in processes and the number of processes and modules in an HDL description. Process decomposition using dataflow analysis methods and predicate abstraction [22] test generation methods are under research now.

## ACKNOWLEDGMENT

Authors would like to thank Russian Foundation for Basic Research (RFBR). The reported study was supported by RFBR, the research project number is 15-07-03834.

## REFERENCES

- [1] J. Bergeron, Writing Testbenches: Functional Verification of HDL Models, Springer, 2003, 478 p.
- [2] V.G.Lazarev and E.I. Piil', Sintez upravlyayushchikh avtomatov (Control automata synthesis), Moscow, Energoatomizdat, 1989, 328 p. (in Russian).
- [3] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking, Cambridge, MIT Press, 2000, 314 p.
- [4] R.J. Ubar, J. Raik, A. Jutman, and M. Jenihhin, "Diagnostic modeling of digital systems with multi-level decision diagrams", Design and Test Technology for Dependable Systems-on-Chip, 2011, pp. 92-118.
- [5] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002), 2009, pp.c1-626.
- [6] IEEE Standard for Verilog Hardware Description Language, IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001), 2006, pp.0\_1-560.
- [7] D. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker", Proceedings of the 16<sup>th</sup> International Conference on Computer Aided Verification (CAV), 2014, № 8559, pp. 334-342.
- [8] D. Deharbe, S. Shankar, and E.M. Clarke, "Model checking VHDL with CV", Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD), 1998, pp. 508-514.
- [9] CBMC model checker. <http://www.cprover.org/cbmc/>
- [10] G. Guglielmo, L. Guglielmo, F. Fummi, and G. Pravadei, "Efficient generation of stimuli for functional verification by backjumping across extended FSMs", Journal of Electronic Functional Testing: Theory and Application, 2011, № 27(2), pp. 137-162.
- [11] I. Melnichenko, A. Kamkin, and S. Smolov, "An extended finite state machine-based approach to code coverage-directed test generation for hardware designs", Proceedings of the Institute for System Programming, 2015, № 27(3), pp. 161-182.
- [12] E. Dijkstra, A Discipline of Programming, Prentice Hall, 1976, 217 p.
- [13] S. Smolov and A. Kamkin, "A method of extended finite state machines construction from HDL descriptions based on static analysis of source code", St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications, № 1(212), 2015, pp. 60-73.
- [14] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin, "Integrating system descriptions by clocked guarded actions, Forum on Design Languages, 2011, pp. 1-8.
- [15] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck "Efficiently computing static single assignment form and the control dependence graph", ACM Transactions on Programming Languages and Systems, № 13(4), 1991, pp. 451-490.
- [16] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, NuXmv 1.0 User Manual. 2014, pp. 7-44. <https://es-static.fbk.eu/tools/nuxmv/index.php?n=Documentation.Home>
- [17] HDL Retrascope toolkit. <http://forge.ispras.ru/projects/retrascope/>
- [18] Fortress library. <http://forge.ispras.ru/projects/solver->
- [19] ITC'99 benchmark. <http://www.cad.polito.it/tools/itc99.html>
- [20] QuestaSim simulator. <https://www.mentor.com/products/fv/questa/>
- [21] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving", Formal Methods in System Design, 2001, Vol. 19 Iss. 1, pp. 7-34.
- [22] E. Clarke, M. Talupur, H. Veith, and D. Wang, "SAT based predicate abstraction for hardware verification", Lecture Notes in Computer Science, 2004, Vol. 2919, pp. 78-92.

# *Deriving adaptive checking sequence for nondeterministic Finite State Machines*

*Anton Ermakov*

*Department of Radiophysics*  
National Research Tomsk State University  
Tomsk, Russia  
antonermak@inbox.ru

*Nina Yevtushenko*

*Department of Radiophysics*  
National Research Tomsk State University  
Tomsk, Russia  
yevtushenko@sibmail.com

**Abstract** — *The derivation of checking sequences for Finite State Machines (FSMs) has a long history. There are many papers devoted to deriving a checking sequence that can distinguish a complete deterministic specification FSM from any non-equivalent FSM with the same number of states. To the best of our knowledge, for nondeterministic FSMs, the topic appeared only recently; the authors started with preset checking sequences for FSMs where the initial state is still known but the reset is very expensive. In this paper, a technique is proposed for deriving an adaptive checking sequence for a complete nondeterministic finite state machine with respect to the reduction relation.*

**Keywords** — *nondeterministic Finite State Machines (FSM), reduction relation, fault model, test derivation, adaptive checking sequences.*

## I. INTRODUCTION

Finite State Machine (FSM) based testing is widely used when deriving conformance tests for interactive discrete event systems [1]; a good example is the FSM based test derivation for telecommunication protocols [2]. In most publications, the specification FSM is assumed to be initialized and a test suite is a set of input sequences which are connected by a reliable reset input [3]. If such reset is rather expensive then so-called checking sequences are used instead of a test suite [4, 5]. For deterministic FSMs such sequences are derived when the specification FSM has a synchronizing (or homing) sequence that takes the FSM from any state to the known state, and a distinguishing sequence that distinguishes every two different states [4]. For nondeterministic FSMs, there are not so many publications on deriving checking sequences, while nondeterministic specifications appear in many applications [5]. One of the reasons for considering nondeterministic specifications is the optionality that is presented in many protocol RFCs [6].

In [5], the authors propose a method for deriving a checking sequence for a complete nondeterministic FSM with respect to the equivalence relation under appropriate limitations on the specification FSM and fault domain. In [7], the authors extend the results for deriving a checking sequence with respect to the reduction relation. A checking sequence is adaptive if the next input depends on the outputs produced to previously applied inputs. Another method for deriving an

adaptive checking sequence with respect to the reduction relation is proposed in [8].

In this paper, we are weakening some limitations of the paper [8] and discuss how to use adaptive synchronizing and distinguishing sequences when deriving an adaptive checking sequence. When deriving preset checking sequences, distinguishing sequences are used while when deriving adaptive tests, it is worth to talk about (adaptive) test cases rather than about distinguishing sequences. The length of an adaptive distinguishing test case can be less than that of a preset distinguishing sequence [9, 10].

In the first part of this paper, we assume that a complete possibly nondeterministic specification FSM has a separating (distinguishing) sequence of reasonable length, each state is deterministically reachable from any other state and an implementation FSM, further called an Implementation Under Test (IUT), is complete and deterministic. Moreover, the behavior of the IUT is not known; we only know the upper bound on the number of states of the IUT. Under the above conditions for the specification FSM, an IUT is a reduction of the specification machine if and only if the IUT is isomorphic to a submachine of the specification FSM [8] and thus, instead of checking IUT traces it is enough to establish the one-to-one correspondence between states and transitions of the specification FSM and the IUT [4]. In other words, each transition of the IUT has to be traversed and a separating sequence has to be applied for verifying the final state of the transition. This approach allows to derive checking sequences of reasonable length when separating and transfer sequence have polynomial length with respect to the number of states of the specification FSM. We briefly sketch how this can be done for an adaptive checking sequence.

In the second part of the paper, we discuss how a separating sequence can be replaced by a distinguishing test case [11, 12] and a simple example illustrates how such replacement can shorten the length of an adaptive checking sequence. Moreover, we note that a distinguishing test case can exist for the specification FSM that has no separating sequence. The rest of the paper is structured as follows. Section II contains the preliminaries. Section III describes an adaptive strategy when using a separating sequence, while Section IV describes the strategy modifications when a distinguishing test case is used instead of a separating

sequence. The conclusions and future work directions are presented in Section V.

## II. PRELIMINARIES

A *finite state machine (FSM)*, or simply a *machine*, is a 4-tuple  $S = \langle S, I, O, h_S \rangle$ , where  $S$  is a finite nonempty set of states,  $I$  and  $O$  are finite input and output alphabets, and  $h_S \subseteq S \times I \times O \times S$  is a (*behavior*) *transition relation*. FSM  $S$  is *non deterministic* if for some pair  $(s, i) \in S \times I$  there can exist several pairs  $(o, s') \in O \times S$  such that  $(s, i, o, s') \in h_S$ . FSM  $S$  is *complete* if for each pair  $(s, i) \in S \times I$  there exists  $(o, s') \in O \times S$  such that  $(s, i, o, s') \in h_S$ . FSM  $S$  is *observable* if for each two transitions  $(s, i, o, s_1), (s, i, o, s_2) \in h_S$  it holds that  $s_1 = s_2$ . FSM  $S$  is *initialized* if it has the designated initial state  $s_1$ , written  $S/s_1$ . Thus, an initialized FSM is a 5-tuple  $\langle S, I, O, h, s_1 \rangle$ . In the following, we consider observable and complete FSMs if the contrary is not explicitly stated. Given FSMs  $S = \langle S, I, O, h, s_1 \rangle$  and  $T = \langle T, I, O, g, t_1 \rangle$ , FSM  $T$  is a submachine  $S$  of if  $T \subseteq S$ ,  $t_1 = s_1$  and  $g \subseteq h$ .

FSM  $S$  is *single-input* if at each state there is at most one defined input at the state, i.e., for each two transitions  $(s, i_1, o_1, s_1), (s, i_2, o_2, s_2) \in h_S$  it holds that  $i_1 = i_2$ , and  $S$  is *output-complete* if for each pair  $(s, i) \in S \times I$  such that the input  $i$  is defined at state  $s$ , there exists a transition from  $s$  with  $i$  for every output in  $O$ . An initialized FSM  $S$  is *acyclic* if the FSM transition diagram has no cycles. An initialized FSM  $S$  is (*initially*) *connected* if each state is reachable from the initial state.

As an example, consider the machine in Fig. 1. The machine is defined over the set of inputs  $I = \{i_1, i_2\}$ , set of outputs  $O = \{0, 1, 2\}$ , and set of states  $S = \{1, 2, 3\}$ . The machine is non-deterministic as for example, from state 2 under the input  $i_2$  there are three outgoing transitions leading to states 2 and 3, respectively.

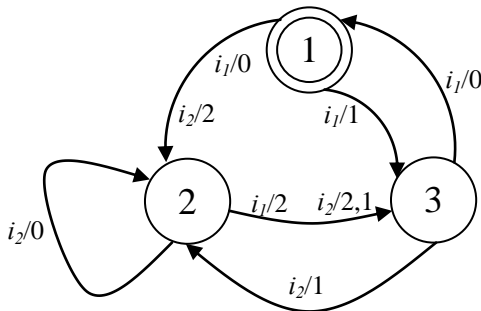


Fig.1. FSM  $S$

Given FSM  $S = \langle S, I, O, h \rangle$ , state  $s$  and an input  $i$ , a sequence of input/output pairs of sequential transitions starting from state  $s$  is a *trace* of  $S$  at state  $s$ ; the set of all traces of  $S$  at state  $s$  including the empty trace is denoted  $Tr(S/s)$ . For state  $s$  and a sequence  $\gamma \in (IO)^*$  of input-output pairs, the  $\gamma$ -*successor* of state  $s$  is the set of all states that are reached from  $s$  by  $\gamma$ . If  $\gamma$  is not a trace at state  $s$  then the  $\gamma$ -successor of state  $s$  is the empty set. For an observable FSM  $S$ , the  $\gamma$ -successor of  $s$  has at most one item. Given an input  $i$  and state  $s$ , state  $s'$  is a

*successor* of state  $s$  under the input  $i$  or simply an *i-successor* of state  $s$  if there exists  $o \in O$  such that the 4-tuple  $(s, i, o, s') \in h$ . Given a subset of states  $M \subseteq S$  and an input  $i$ , the set of states  $M'$  that is the union of  $i$ -successors over states of the set  $M$ , is a *successor* of the set  $M$  under the input  $i$  or simply an *i-successor* of  $M$ . As an example, in Fig. 1, the  $i_2$ -successor of state 2 is the set  $\{2, 3\}$  and the  $i_2$ -successor of state 1 is  $\{2\}$ . Thus, the  $i_2$ -successor of the set of states  $\{1, 2\}$  is the set  $\{2, 3\}$ .

In a usual way, the behavior relation is extended to input and output sequences. Given states  $s$  and  $s'$ , the defined input sequence  $\alpha$  can *take* (or simply *takes*) the FSM  $S$  from state  $s$  to state  $s'$  if there exists an output sequence  $\beta$  such that  $(s, \alpha, \beta, s') \in h$ . The notation  $successor(s, \alpha)$  is used for denoting the set of all states reachable from state  $s$  after applying the defined input sequence  $\alpha$ , i.e.,  $successor(s, \alpha) = \{s' : \exists \beta \in O^* [(s, \alpha, \beta, s') \in h]\}$ . The set  $out(s, \alpha)$  denotes the set of all output sequences (responses) that the FSM  $S$  can produce at state  $s$  in response to a defined input sequence  $\alpha$ , i.e.,  $out(s, \alpha) = \{\beta : \exists s' \in S [(s, \alpha, \beta, s') \in h]\}$ . For example, using the machine in Fig. 1,  $successor(2, i_2 i_1) = \{1, 3\}$  and  $out(2, i_2 i_1) = \{02, 20, 10\}$ . Given states  $s$  and  $s'$ , input sequence  $\alpha$  and output sequence  $\beta$  of the same length, the trace  $\alpha/\beta$  can *take* (or simply *takes*) the FSM  $S$  from state  $s$  to state  $s'$  if  $(s, \alpha, \beta, s') \in h$ . An FSM  $S$  is *deterministically connected (d-connected)* if for each pair  $s, s' \in S$  there exists an input sequence  $\alpha$  such that  $successor(s, \alpha) = \{s'\}$ ; in this case, we say that  $s'$  is *d-reachable* from state  $s$ . The input sequence such that  $successor(s, \alpha) = \{s'\}$  is denoted  $\alpha_{ss'}$ . An initialized machine  $S$  is *connected* if for each  $s \in S$  there exists an input sequence that takes the FSM  $S$  from the initial state to state  $s$ . Given initialized FSMs  $S = \langle S, I, O, h, s_1 \rangle$  and  $P = \langle P, I, O, g, p_1 \rangle$ , the *intersection*  $S \cap P$  is the largest connected submachine of FSM  $\langle S \times P, I, O, f, s_1 p_1 \rangle$  where  $(sp, i, o, s'p') \in f \Leftrightarrow (s, i, o, s') \in h \ \& \ (p, i, o, p') \in g$ . Given a complete FSM  $S = \langle S, I, O, h \rangle$ , states  $s_1$  and  $s_2$  of  $S$  are *non-separable* if for each input sequence  $\alpha \in I^*$  it holds that  $out(s_1, \alpha) \cap out(s_2, \alpha) \neq \emptyset$ , i.e., the sets of output responses at states  $s_1$  and  $s_2$  to each input sequence intersect; otherwise, states  $s_1$  and  $s_2$  are *separable*. For separable states  $s_1$  and  $s_2$ , there exists an input sequence  $\alpha \in I^*$  such that  $out(s_1, \alpha) \cap out(s_2, \alpha) = \emptyset$ , i.e., the sets of output responses at states  $s_1$  and  $s_2$  to the input sequence  $\alpha$  are disjoint. In this case,  $\alpha$  is a *separating* sequence of states  $s_1$  and  $s_2$ , or simply *separates*  $s_1$  and  $s_2$ , written  $s \star_\alpha p$ . As an example, consider states 1 and 2 of the machine in Fig. 1. The sets of output sequences produced by  $S$  at states 1 and 2 to the input sequence  $i_2 i_1$  are  $\{22\}$  and  $\{02, 20, 10\}$  which are disjoint; thus, the sequence  $i_2 i_1$  is a separating sequence for these two states. If a sequence  $\alpha$  separates every pair of states of FSM  $S$  then is a *separating sequence* for FSM  $S$ . By direct inspection, one can assure that  $i_2 i_1$  separates each pair of different states of the FSM  $S$  as the set of output sequences produced by  $S$  to  $i_2 i_1$  at state 3 is  $\{12\}$ . Methods for checking the existence of a separating sequence together with its derivation (when such a sequence exists) can be found in [13]. Unfortunately, it is known that the length of a separating sequence can be exponential with respect to the number of states of FSM  $S$ .

despite the fact that performed experiments show that generally, if a separating sequence exists then it is rather short [14]. Given complete FSMs  $S$  and  $P$ , state  $p$  of the FSM  $P$  is a *reduction* of state  $s$  of the FSM  $S$ , written  $p \leq s$ , if the set of traces of  $P$  at state  $p$  is a subset of that of  $S$  at state  $s$ ; otherwise,  $p$  is not a reduction of state  $s$ , written  $p \not\leq s$ . The initialized FSM  $P/p_1$  is a reduction of initialized FSM  $S/s_1$  if  $p_1 \leq s_1$ , i.e., if the set of traces of  $P/p_1$  is a subset of that of  $S/s_1$ . If the sets of traces of the FSMs  $S/s_1$  and  $P/p_1$  coincide then these machines are *equivalent* [11].

**Fault model.** When using FSM based derivation, it is assumed that the specification FSM describes the reference behavior while the fault domain contains each possible FSM implementation of the specification FSM. In our case, all the machines are initialized complete and observable; moreover, an implementation FSM is assumed to be deterministic. As a conformance relation we consider the reduction relation. In other words, we implicitly assume that the nondeterminism of the specification is implied by the optionality where a designer selects a better option according to some criteria. There still is a reliable reset but it is rather expensive and can be used only once.

Correspondingly, we consider a fault model  $FM = \langle S/s_1, \leq, \Omega \rangle$  where  $S/s_1$  is a complete possibly nondeterministic observable initialized FSM with  $n$  states,  $n > 1$ ,  $\Omega$  is the set of all complete deterministic FSMs over the same input alphabet as  $S$  with at most  $n$  states.

An adaptive strategy for testing a given FSM is a procedure that derives the next input based on the output of the IUT to the previous inputs. An adaptive strategy is *complete* with respect to the fault model  $FM$  if for each  $P/p_1 \in \Omega$ , the output to the applied input sequence is contained in the set of specification output responses if and only if FSM  $P/p_1$  is a reduction of  $S/s_1$ . The following two propositions can be useful when proving that a proposed adaptive strategy is complete with respect to the given  $FM$ .

**Proposition 1 [11].** Given complete observable FSMs  $S$  and  $P$ , the initialized FSM  $P/p_1$  is a reduction of initialized FSM  $S/s_1$  if and only if the intersection  $P/p_1 \cap S/s_1$  is a complete FSM.

**Proposition 2 [15].** Given complete observable FSMs  $P/p_1$  and  $S/s_1$ , let  $S$  have a separating sequence and be  $d$ -connected. FSM  $P/p_1$  is a reduction of  $S/s_1$  if and only if  $P/p_1$  is isomorphic to some submachine of  $S/s_1$ .

Proposition 2 shows a way how a complete adaptive strategy can be elaborated. Each transition of an IUT has to be traversed and the final state should be checked by the use of a separating sequence. For this reason, the procedure is divided into two steps. At the first step, we check that an IUT has exactly  $n$  states and fix the output response to a separating sequence at each state. At the second step, a transition at each state of the IUT under each input is traversed and the final state of the transition is checked by the use of a separating sequence.

**Example 1.** Consider the specification FSM in Fig. 1. The FSM  $S$  has a separating sequence  $i_2 i_1$ . Table I has output responses to this sequence at each state.

TABLE I. OUTPUT RESPONSES TO  $i_2 i_1$

State	Outputs for separate sequences: $i_2 i_1$
1	22
2	02, 20, 10
3	12

Moreover, the FSM in Fig. 1 is  $d$ -connected, i.e., for each pair  $j$  and  $k$  of different states there exists a  $d$ -transfer sequence  $\alpha_{jk}$ :  $\alpha_{12} = i_2$ ,  $\alpha_{23} = i_1$ , и  $\alpha_{31} = i_1$ .

### III. ADAPTIVE STRATEGY FOR DERIVING A CHECKING SEQUENCE

Here we briefly follow the adaptive strategy proposed in [8].

**Input.** FSM  $S = (S, I, O, h_s, s_1)$  with  $n$  states, a separating sequence  $\delta$  for FSM  $S$ ,  $d$ -transfer sequences  $\alpha_{ss'}$  for each pair of different states  $s$  and  $s'$ , a complete deterministic implementation  $P/p_1$  with at most  $n$  states.

**Output.** The messages ‘ $P/p_1$  is a reduction of  $S/s_1$ ’ or ‘ $P/p_1$  is not a reduction of  $S/s_1$ ’ and an input sequence  $\sigma$  that  $r$ -distinguishes  $P/p_1$  from  $S/s_1$  in the latter case.

A procedure is divided in two steps. At the first step, each state is  $d$ -reached in  $S/s_1$ , and checked by the use of  $\delta$  whether a state reached in  $P/p_1$  corresponds to this state. The output responses and corresponding successors are saved in the set *Separable*.

The set *Separable* has triples  $(s, \rho, s')$  where  $s$  is a current state of  $S$ ,  $\rho = out_P(s, \delta)$  is the output response of  $P$  to the separating sequence  $\delta$  at state that corresponds to state  $s$  and  $s'$  is the  $\delta/\rho$ -successor of state  $s$ . The procedure is simple enough: we apply a separating sequence until new states of the FSM  $S$  appear as  $\delta/\rho$ -successors. Once a  $\delta/\rho$ -successor is one of already checked states, the  $d$ -transfer sequence to a non-traversed state  $s$  is applied and the procedure of applying  $\delta$  is repeated. Once an unexpected output response is obtained the procedure is terminated and there appears the message that  $P$  is not a reduction of  $S$ .

**Example 1.** Suppose that the FSM  $S$  in Fig. 1 is the specification FSM while the FSM  $P$  in Fig. 2 is an implementation FSM at hand, i.e.,  $P$  is an IUT.

A separating sequence  $\delta = i_2 i_1$ . At the initial state we obtain 22 as the output response to  $\delta$ , and thus, the initial state  $a$  of the IUT corresponds to the initial state  $s_1$  of the specification FSM  $S$ . After applying  $\delta$  one more time the output response 12 is obtained, i.e., the state 3 of the FSM  $S$  reached after  $\delta/12$  corresponds to state  $c$  of the IUT. We apply  $\delta$  one more time and obtain 12 again, i.e., we reach a state that corresponds to state 3. After applying  $\alpha_{32} = i_2$ , state 2 is reached in the specification FSM  $S$  and after applying  $\delta = i_2 i_1$  twice we realize that there is a transition from a new state  $b$

under  $\delta$  to state  $a$ . Since  $P$  has at most three states, all of them are identified by  $\delta$  as  $\delta$  is a separating sequence for the IUT. Thus,  $Separate = \{(1, 22, 3), (2, 10, 1), (3, 12, 3)\}$  where for each triple, the first item is a current state, the next item is the IUT response at the corresponding state while the last item indicates the next IUT state under the separating sequence. In other words, we have established one-to-one correspondence between states of  $S$  and  $P$ : 1 and  $a$ , 2 and  $b$ , 3 and  $c$ .

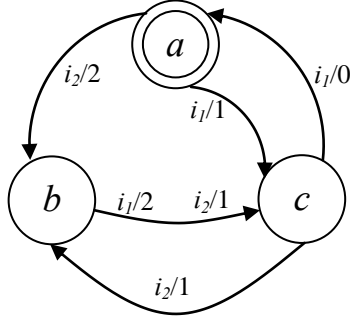


Fig. 2. An implementation FSM  $P$  of the specification machine in Fig. 1

At the next step, the set *Transition* is constructed. This set contains 4-tuples  $(s, i, o, s')$  and the derivation is terminated once there exists a 4-tuple for each pair  $(s, i) \in S \times I$ . For this purpose, for each state, each input followed by the separating sequence has to be applied; after this, a corresponding 4-tuple is added to the set *Transition*. Once at the current state all the pairs  $(s, i) \in S \times I$  are already checked, the sequence of transitions of the set *Transition* is used in order to reach a state with an unchecked transition. Such a sequence exists, since according to the specification FSM features, a reduction of  $S$  is isomorphic to a submachine of  $S$  with  $n$  states (Proposition 2), i.e., is strongly connected according to the FSM  $S$  features. The first part of an applied sequence confirms that an implementation under test has exactly  $n$  states and there is one-to-one correspondence between states of the specification and implementation FSMs. The second part confirms the one-to-one correspondence between transitions of these machines. Thus, if no unexpected output is produced then and only then an implementation under test is a reduction of the specification FSM.

**Example 1.** Table II represents the results obtained after applying the first part of checking sequence.

TABLE II. THE SET *SEPARATE* FOR THE FSM  $P$

$s$	$\rho$	$s'$
$a(1)$	22	$c(3)$
$b(2)$	10	$a(1)$
$c(3)$	12	$c(3)$

Moreover, we already have a current checking sequence  $\sigma = i_2 i_1 i_2 i_1 i_2 i_2 i_1 i_2 i_1$  that terminates at state  $c$  that corresponds to state 3 of the FSM  $S$ .

The set *Transition* is empty and there is an unchecked transition at state 3 under input  $i_1$ . After applying  $i_1$  to the IUT  $P$ ,  $P$  produces the expected output 1, while after applying  $\delta$  we obtain 12 which corresponds to Line 3 of Table II, i.e., we

conclude that the IUT reaches a state  $c$  that corresponds to state 3. At this state there is an unchecked input  $i_2$ . After applying this input, the IUT  $P$  produces the expected output 1 while after applying  $\delta$  we obtain 22 which corresponds to Line 3 of Table II, i.e., we conclude that the IUT reaches a state  $a$  that corresponds to state 1. Other transitions of  $P$  are checked in the same way. As a result, we have a checking sequence  $\sigma = i_2 i_1 i_2 i_1 i_2 i_2 i_1 i_2 i_1 + i_1 i_2 i_1 i_2 i_1 i_1 i_2 i_1 i_2 i_1 i_2 i_2 i_1$ , for which the output response is contained in the set of output responses of  $S$  to this sequence and thus, we conclude that the IUT in Fig. 2 is a reduction of the FSM  $S$  in Fig. 1.

#### IV. USING (ADAPTIVE) DISTINGUISHING TEST CASES INSTEAD OF SEPARATING SEQUENCE

Consider an adaptive strategy presented in Section III. There are hard limitations imposed for the specification FSM: there has to exist a separating sequence as well as  $d$ -transfer sequences. However, not each FSM possesses these features and moreover, generally, the length of these sequences can be exponential w.r.t the number of states [11].

For this reason, we weaken our restrictions for the specification FSM using not a separating sequence but a distinguishing test case which model an adaptive distinguishing experiment with an IUT at hand [13]. First, it is known that a distinguishing test case can exist for FSMs which do not possess a separating sequence. Second, the height of the test case (the length of an adaptive distinguishing sequence) generally is shorter than that of a separating sequence [9]. We then briefly define the notion of a (adaptive) distinguishing test case.

Given an input alphabet  $I$  and an output alphabet  $O$ , a *test case*  $TC(I, O)$  is an initially connected single-input output-complete observable initialized FSM with the acyclic transition graph. By definition, if  $|I| > 1$  then a test case is a partial FSM. A test case  $TC(I, O)$  over alphabets  $I$  and  $O$  defines an adaptive experiment with any FSM over alphabets  $I$  and  $O$ .

In general, given a test case  $TC$ , *length* of the test case  $TC$  is defined as the length of the longest trace from the initial state to a deadlock state of  $TC$  and it specifies the length of the longest input sequence that can be applied to an FSM  $S$  during the adaptive experiment that is described by the test case; this length is often called the *height* of the adaptive experiment. As usual, for testing, one is interested in deriving a test case (experiment) with minimal length (height). It is known that if the specification FSM is complete and merging-free, i.e., for each input  $i$  and output  $o$ , the non-empty  $io$ -successors of two different states do not coincide, then the length of an adaptive distinguishing test case (if it exists) is polynomial w.r.t. the number of states of the FSM. In fact, if  $S$  has  $n$  states then the length of an adaptive distinguishing test case (if it exists) is of the order  $O(n^3)$  [16]. The class of merging-free FSMs is big enough; at least it contains many deterministic FSMs which are used in practical applications [17].

Given a complete observable FSM  $S$  over input and output alphabets  $I$  and  $O$ , let  $TC$  be a test case over alphabets  $I$  and  $O$ . A test case  $TC$  is a *distinguishing test case* for FSM  $S$  if for



each deadlock state of  $TC$ , it holds that a trace labeling the path from the root to this state is a trace at most at a single state of the FSM  $S$ .

**Example 2.** As an example, consider an FSM from [12]. By direct inspection, one can assure that there is no separating sequence; however, states are pairwise  $d$ -reachable and there exists a distinguishing test case shown in Fig. 4. The deadlock states are labeled with a corresponding initial state of the FSM  $S$ . Thus, when deriving a checking sequence, this distinguishing test case can be used instead of separating sequence. The only difference is that when constructing the set *Separate* (Table II) we will get not the output responses at all states to a single separating sequence but will ensure that all the states of the IUT have corresponding state identifiers.

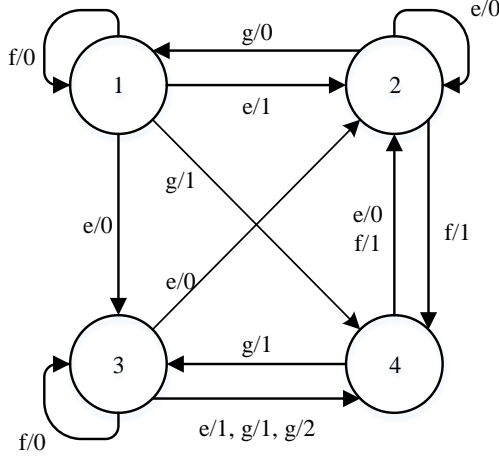


Fig. 3. Specification FSM  $S$

Moreover, the FSM in Fig. 3 is  $d$ -connected; for each pair  $j$  and  $k$  of different states there exists a  $d$ -transfer sequence  $\alpha_{jk}$ :  $\alpha_{12} = ge$ ,  $\alpha_{13} = gg$ ,  $\alpha_{14} = g$ ,  $\alpha_{21} = g$ ,  $\alpha_{23} = fg$ ,  $\alpha_{24} = f$ ,  $\alpha_{31} = gfg$ ,  $\alpha_{32} = gf$ ,  $\alpha_{34} = g$ ,  $\alpha_{41} = eg$ ,  $\alpha_{42} = e$ ,  $\alpha_{43} = g$ .

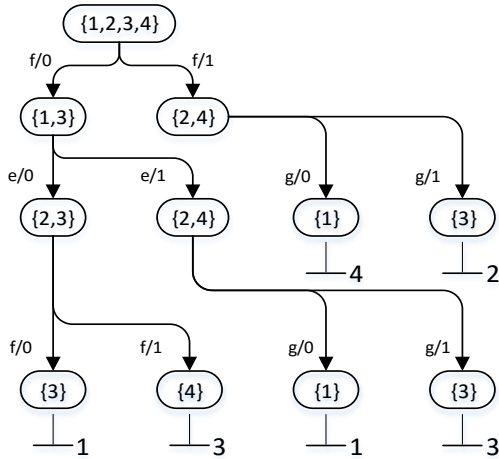


Fig. 4. A distinguishing test case  $T$  for the FSM  $S$  in Fig. 3

Given a distinguishing test case  $TC$  for FSM  $S$ , a trace from the initial state to a deadlock state is a *complete* trace and  $Complete(TC)$  is the set of all complete traces of  $TC$ .

Given FSM  $S$  and an input sequence  $\alpha$ ,  $\alpha$  is a *state identifier* of state  $s$  of FSM  $S$  if  $\alpha$  is a separating sequence for each pair  $(s, s')$  where  $s' \neq s$ . At states 1 and 3 of FSM  $S$  in Fig. 3, sequences  $fef$  and  $feg$  are state identifiers while at states 2 and 4, a sequence  $fg$  is a state identifier. Given the specification FSM  $S$  that has a distinguishing test case  $TC$ , let  $P$  be a deterministic complete FSM with the same number of states. The FSM  $P$  is  $TC$ -compatible with  $S$  if there exists one-to-one correspondence  $F: S \rightarrow P$  such that for each state  $s \in S$  it holds that the intersection of  $Tr(S/s) \cap Tr(P/p) \cap Complete(TC)$  is not empty if and only if  $p = F(s)$ .

**Theorem 3.** Given the specification FSM  $S$  that has a distinguishing test case  $TC$ , let a deterministic complete FSM  $P$  be  $TC$ -compatible with  $S$ . For each state  $p$  of  $P$ , the distinguishing test case  $TC$  has a complete trace  $\alpha/\beta$  that is a trace at state  $p$ ; moreover,  $\alpha$  is a state identifier of state  $p$  in  $P$ .

In fact, if there exists one-to-one correspondence between states of  $S$  and  $P$  according to the distinguishing test case  $TC$ , then for each two states  $s$  and  $s'$ ,  $s' \neq s$ , there exists a prefix of an input sequence of some complete trace of  $TC$  such that output responses at corresponding states  $p = F(s)$  and  $p' = F(s')$  are different. As  $TC$  is a distinguishing test case of  $S$  and  $P$  is complete and deterministic, the latter means that a corresponding input projection of trace  $\alpha/\beta$  is a state identifier of state  $p$ . According to Theorem 3, an adaptive distinguishing sequence can be derived in the same way as when using a separating sequence. The only difference is that at the first step, for each state, the set *Separate* contains a corresponding state identifier and the next state.

**Example 2.** Let an IUT be an FSM  $P$  in Fig. 5. By direct inspection, one can assure that  $P$  is isomorphic to a submachine of  $S$ , i.e., is a reduction of FSM  $S$  in Fig. 3. Below we illustrate an adaptive strategy that allows drawing this conclusion and the length of a corresponding checking sequence.

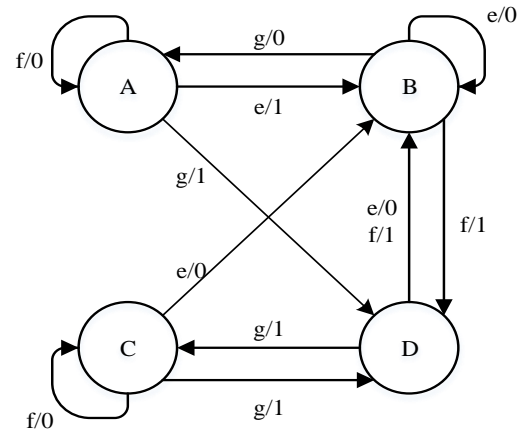


Fig. 5. An implementation FSM  $P$

We apply an adaptive strategy in order to check whether  $P$  (Fig. 5) is a reduction of  $S$  (Fig. 3). Starting at the initial state  $A$ , we apply  $feg$  that is a state identifier of the initial state 1 of FSM  $S$  according to the distinguishing test case (Fig. 4). The IUT produces the response 010 and we conclude that state  $A$  of  $P$  corresponds to state 1 of  $S$ . Moreover, in order to find out

where the input sequence  $feg$  takes the IUT we apply  $feg$  again, obtain 010 and conclude that  $feg$  takes the IUT to state  $A$  that corresponds to state 1 of the specification FSM  $S$ .

As we already know that state  $A$  corresponds to state 1, we apply a  $d$ -transfer sequence  $\alpha_{12} = ge$ , the IUT produces the response 10 and moves to a new state  $B$  of  $P$  that corresponds to state 2 of  $S$ . In order to verify this, we apply a state identifier  $fg$  of state 2 and obtain the output response 11, i.e., state  $B$  of  $P$  corresponds to state 2 of  $S$ . The IUT moves to state  $C$  that should correspond to state 3 of  $S$ . We apply the sequence  $fef$ , obtain the response 001 and the IUT reaches a new state  $D$  that should correspond to state 4 of  $S$ . To verify this, we apply a state identifier  $fg$ , obtain the response 10 and the IUT moves to the initial state  $A$ .

The application of  $feg$  that is a state identifier of  $A$  terminates the procedure and Table III is obtained. Since the IUT has at most four states this table contains a state identifier for each state, the corresponding output response and the next state of the IUT.

TABLE III. THE SET *SEPARATE* FOR THE FSM IN FIG. 3 ACCORDING TO THE TEST CASE IN FIG. 4

Current state	State identifier	Output response	Next state
A (1)	$feg$	010	A (1)
B (2)	$fg$	11	C (3)
C (3)	$fef$	001	D (4)
D (4)	$fg$	10	A (1)

The set *Transition* is derived similar to that when using a separating sequence, i.e., after applying each input at each state the next state is verified according to the known state identifier (Table III). If at a current state there are no unchecked transitions then using the already constructed part of the set *Transition* we reach a state where are unchecked inputs. In our running example for the specification in Fig. 3 and an IUT in Fig. 5, we have obtained a checking sequence of length 68 and since only expected output responses have been obtained from the IUT, we have concluded that the IUT is a reduction of the specification FSM  $S$ .

## V. CONCLUSIONS

In this paper, we have proposed an adaptive strategy for testing a deterministic implementation FSM with respect to the nondeterministic specification FSM and reduction relation when the upper bound on the number of the implementation FSM states is known. Similar to deterministic FSMs, the strategy can be applied under appropriate restrictions upon the specification FSM and fault domain. However, we show that the requirement of the existence of a distinguishing (separating) sequence can be replaced by the requirement of the existence of a distinguishing test case. The latter can exist when there is no separating sequence and usually length of a distinguishing test case is less than that of a separating sequence (if both exist). Moreover, using adaptive transfer sequences [18] we can weaken the requirement for the existence of  $d$ -transfer sequences. We also can use adaptive

synchronizing sequences instead of the reset and in this case, an adaptive strategy can be applied for non-initialized FSMs and as a future work, we are going to develop such a strategy.

**Acknowledgement.** This work is partly supported by RFBR grant No. 15-58-46013 CT\_a.

## REFERENCES

- [1] Z. Kohavi. Switching and Finite Automata Theory, McGraw-Hill, New York, 1978.
- [2] T.S. Chow. "Testing software Design Modelled by Finite State Machines", In IEEE Trans. Software Eng. Vol. 4 (3), 1978, pp. 178-187.
- [3] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, N. Yevtushenko. "FSM-based conformance testing methods: A survey annotated with experimental evaluation", In Information & Software Technology, Vol. 52 (12), 2010, pp. 1286-1297.
- [4] F.C. Hennie. "Fault-Detecting Experiments for Sequential Circuits", In Proc. Fifth Ann. Symp. Switching Circuit Theory and Logical Design, 1964, pp. 95-110.
- [5] A. Petrenko, A. Simão, N. Yevtushenko. "Generating Checking Sequences for Nondeterministic Finite State Machines", In Proceedings of the ICST, 2012, pp. 310-319.
- [6] M. Zhigulin, A. Kolomeez, N. Kushik, A. Shabaldin... "EFSM based testing a software implementation of IRC protocol", In Izvestia Tomskogo polytechnicheskogo instituta, 318 (5), 2011, pp. 81-84 (in Russian).
- [7] A. Petrenko, A. Simão. "Generalizing the DS-Methods for Testing Non-Deterministic FSMs", In Comput. J. 58(7), 2015, pp. 1656-1672.
- [8] A. Ermakov. "Deriving checking sequences for nondeterministic FSMs", In Proceedings of the Institute for System Programming of RAS, Vol. 26, 2014, pp. 111-124 (in Russian).
- [9] R. Alur, C. Courcoubetis, M. Yannakakis. "Distinguishing tests for nondeterministic and probabilistic machines", In Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, 1995, pp. 363-372.
- [10] N. Kushik, K. El-Fakih, N. Yevtushenko. "Adaptive Homing and Distinguishing Experiments for Nondeterministic Finite State Machines", In Lecture Notes in Computer Science, Vol. 8254, 2013, pp. 33-48.
- [11] A. Petrenko, N. Yevtushenko. "Conformance Tests as Checking Experiments for Partial Nondeterministic FSM", In Lecture Notes in Computer Science, Vol. 3997, 2005, pp. 118-133.
- [12] A. Petrenko, N. Yevtushenko. "Adaptive Testing of Deterministic Implementations Specified by Nondeterministic FSMs", In Lecture Notes in Computer Science, Vol. 7019, 2011, pp. 162-178.
- [13] N. Kushik. Methods for deriving homing and distinguishing experiments for nondeterministic FSMs. PhD thesis, Tomsk State University, 2013 (in Russian).
- [14] N. Shabaldina, K. El-Fakih, N. Yevtushenko. "Testing Nondeterministic Finite State Machines with Respect to the Separability Relation", In Proceedings of Intern. Conf. on Testing Systems and Software (ICTSS/FATYES), 2007, pp. 305-318.
- [15] M. Vetrova. FSM based methods for compensator design and testing. PhD thesis, Tomsk State University, 2004 (in Russian).
- [16] N. Yevtushenko, N. Kushik. Decreasing the length of Adaptive Distinguishing Experiments for Nondeterministic Merging-free Finite State Machines // Proceedings of IEEE East-West Design & Test Symposium, pp.338 – 341.
- [17] C. Güniçen, K. Inan, U.C. Türker, H. Yenigün. "The relation between preset distinguishing sequences and synchronizing sequences", In Formal Aspects of Computing, Vol. 26 (6), 2014, pp. 1153-1167.
- [18] N. Kushik, N. Yevtushenko, H. Yenigün. "Reducing the complexity of checking the existence and derivation of adaptive synchronizing experiments for nondeterministic FSMs", In Proceedings of International Workshop on Domain Specific Model-based Approaches to Verification and Validation (AMARETTO'2016), 2016, pp. 83-90.

# *Conversion of abstract behavioral scenarios into scenarios applicable for testing*

Pavel Drobintsev, Vsevolod Kotlyarov, Igor Nikiforov, Nikita Voinov, Ivan Selin

Institute of Computer Science and Control  
Peter the Great Saint Petersburg Polytechnic University  
Saint Petersburg, Russia  
drob@ics2.ecd.spbstu.ru

**Abstract**—In this article an approach of detailing verified test scenarios without losing the model's semantics is proposed. Process of translating abstract data structures into detailed data structures used in system implementation is presented with examples.

**Keywords**—model approach; model verification; test mapping;

## I. INTRODUCTION

One of the most perspective approaches to modern software product creation is usage of model oriented technologies both for software development and testing. Such technologies are called MDA (Model Driven Architecture) [1,2], MDD (Model Driven Development) [2] and MDSD (Model Driven Software Development) [3]. All of them are mainly aimed to design and generation of application target code based on a formal model.

The article is devoted to specifics of model oriented approaches usage in design and generation of large industrial software applications. These applications are characterized by multilevel representation related to detailing application functionality to the level where correct code is directly generated.

The idea of model oriented approach is in creating of multilevel model of application during design process. This model is iteratively specified and detailed to the level when executable code can be generated. On the design stage formal model specification allows using verification together with other methods of static analysis with goal to guaranty correctness of the model on early stages of application development.

More than 80% [4] of model-oriented approaches are using graphical notations, which allows simplifying of work with formal notations for developers. Requirements for knowledge of testers and customer representatives is reduced by this way and process of models developing are also simplified.

## II. LEVELS OF BEHAVIORAL MODELS DEVELOPMENT

One of high level languages for system formal model specification is Use Case Maps (UCM) [5, 6]. It provides visible and easy understandable graphical notation. Further abstract models will be specified in UCM language to demonstrate proposed approach in details. Also considered is VRS/TAT technology chain [7], which uses formal UCM models for behavioral scenarios generation.

Traditional steps of formal abstract model development in UCM language are the following:

1. Specifying main interacting agents (components) and their properties, attributes set by agent and global variables.
2. Introducing main system behaviors to the model and developing diagrams of agent's interaction control flow.
3. Developing internal behaviors for each agent and specifying data flow in the system.

Undoubted benefit of UCM language is possibility to create detailed structured behavioral diagrams. Structuring is specified both by Stub structural elements and reused diagrams (Maps), which are modeling function calls or macro substitution. Unfortunately, standard UCM language deals with primitive and abstract data structures, which are not enough to check implementation of a real system. This drawback is compensated by using metadata mechanism [6]. But metadata does not allow detailing data flow to more detailed levels. That's why for creating detailed behaviors it is proposed to use vertical levels of abstractions during behavioral models development which are: structured system model in UCM language, behavioral scenarios with symbolic values and variables, concrete behavioral scenarios are behavioral scenarios with detailed data structures.

Another benefit of UCM usage is possibility to execute model verification process. UCM diagrams are used as input for VRS/TAT toolset which provides checks for specifications correctness. These checks can detect issues with unreachable states in the model, uninitialized variables in metadata, counterexamples for definite path in UCM, etc. After all checks are completed the user gets a verdict with a list of all findings and a set of counterexamples which show those paths in UCM model which lead to issue situations. If a finding is considered to be an error, the model is corrected and verification process is launched again. As a result after all fixes a correct formal model is obtained which can be used for further generation of test scenarios.

After formal model of a system has been specified in UCM language, behavioral scenarios generation is performed. Note that behavioral generator is based not on concrete values assigned to global variables and agents attributes, but on symbolic ones which reduces significantly the number of behavioral scenarios covering the model. However symbolic

test scenarios cannot be used for applications testing as executing behavioral scenarios on the real system requires concrete values for variables. So the problem of different level of abstraction between model and real system still exists. In VRS/TAT technology concretization step [8] is used to convert symbolic test scenarios. On this step ranges of possible values for variables and attributes are calculated based on symbolic formula and symbolic values are substituted with concrete ones. But concretization of abstract model's behavioral scenarios is not enough for their execution, because on this stage scenarios still use abstract data structures which differ from data structures in real system. As a result conversion of concretized behavioral scenarios of abstract UCM level into scenarios of real system level was integrated into technology chain for behavioral scenarios generation

### III. CONCRETIZATION

In behavioral scenarios data structures are mainly used in signals parameters. There are two types of signals in UCM model: incoming to an agent and outgoing from an agent. Incoming signals are specified with the keyword "in" and can be sent either by an agent or from outside the system specifying with the keyword "found". Outgoing signals are specified with the keyword "out" and can be sent either to an agent or to outside the system specifying with the keyword "lost".

An example of outgoing signal can be seen on Fig. 1. The element "send\_Fwd\_Rel\_Req\_V2\_papu" contains metadata with the signal "Forward\_Relocation\_Request\_V2" and UCM-level parameter "no\_dns". Outgoing signals can only be used inside of "do" section as a reaction of the system on some event.

Another example is the element "rev\_Rel\_RSP\_papu", which contains metadata with the incoming signal "Forward\_Relocation\_Response\_V2" and UCM-level parameters "seq\_nbr", "cause\_dec", "cs" "reccod" (Fig. 2).

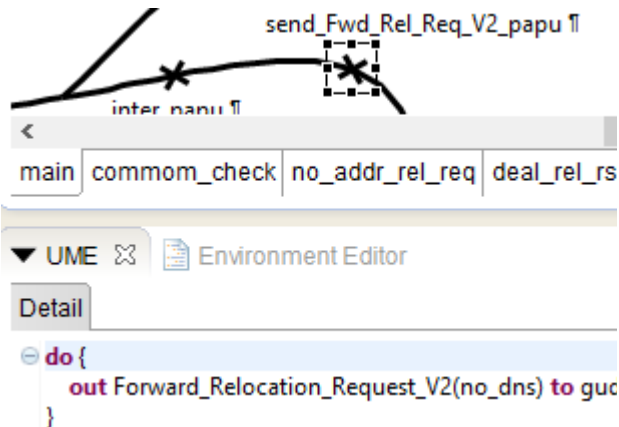


Fig. 1. Description of "Forward\_Relocation\_Request\_V2" signal in metadata corresponding UCM element

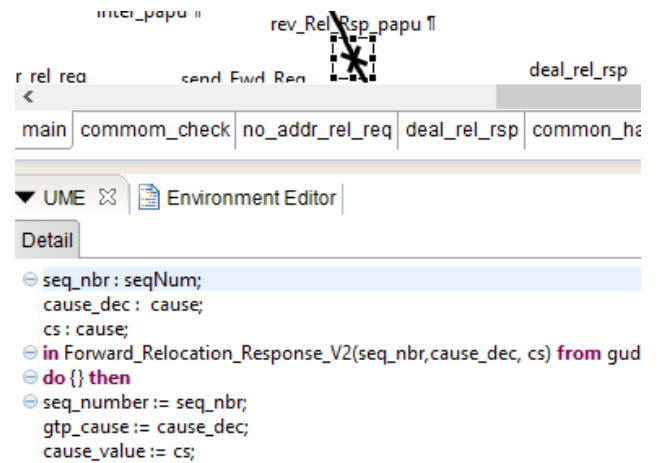


Fig. 2. Description of the "Forward\_Relocation\_Response\_V2" signal in metadata of the "rev\_Rel\_RSP\_papu" UCM element

If the signal Forward\_Relocation\_Response\_V2 is received, then new values taken from signal parameters are assigned to variables.

Consider an example of converting signal structure of UCM level into detailed structures of real system for the signal "gtp\_forward\_relocation\_req\_s". Based on high level UCM model symbolic behavioral scenarios are generated containing data structures described in metadata of UCM elements. Fig.3 contains symbolic test scenario where the agent "GTP#gtp" receives the signal "gtp\_forward\_relocation\_req\_s" from agent "GMG#gmg". In symbolic scenarios actual names of UCM model agents specified in metadata are used.

Symbolic behavioral scenario is input data for concretization module, which substitutes symbolic parameters with concrete values. In current example the parameters "seq\_nbr", "ip1", "ip2", "tid" and "isIntra" are substituted with values "invalid", "valid", "exist", "valid" and "0". Fig.4 contains concrete behavioral scenario.

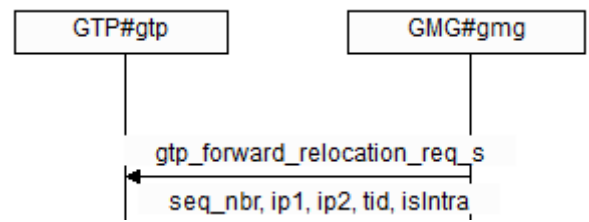


Fig. 3. Symbolic test scenario with the signal "gtp\_forward\_relocation\_req\_s"

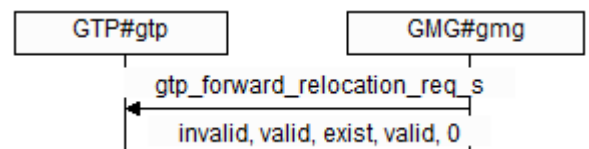


Fig. 4. Concrete test scenario with the signal "gtp\_forward\_relocation\_req\_s"

#### IV. DATA STRUCTURES CONVERSION

After concretization, scenarios still have to be processed because their structure does not match with one's of system under test (SUT). The most straightforward approach is to manually review all generated scenarios and edit all used signals so that their structure will reflect SUT interfaces. Obviously, it will require too much time and may be a bottleneck of the whole process. Therefore, there is a need for automation.

The common way is making a wrapper that transforms signals to desired form using one of popular programming languages (C++, Java, etc.). However, it could lead to making new mistakes and loss of correctness of test scenarios. The main reason for this is ability to implement incorrect structures on scenarios level. In addition, other language-specific errors are possible. Cutting down the ability to produce incorrect code will reduce the number of mistakes while still maintaining good level of automation.

##### A. Approach

To be able to satisfy these needs a two-step approach called "Lowering" was suggested. The name comes from descending on lower levels of abstraction. In general, lowering can be described as creating processing rules for each signal called "lowering rules" and application of these rules to the concrete scenarios.

As said above, there are some restrictions on possible operations to save the correctness of test scenarios, such as:

- It is prohibited to separate constants into several independent parts (e.g. separating value 1536 in 15 and 36 is not possible)

```
LoweringSpec ::= UCMSignal "->" LoweringRule |
LoweringSpec UCMSignal "->" LoweringRule
LoweringRule ::= LoweringCondition | LoweringRule
LoweringCondition
LoweringCondition ::= <condition STRING>
ConditionContent
ConditionContent ::= LoweredElement | LoweredElement
ConditionContent
LoweredElement ::= LoweredDo | LoweredSignal |
LoweredAction
LoweredDo ::= <code STRING>
LoweringSignal ::= <signal name STRING>
SignalContent
SignalContent ::= ValueNotation Instance Via
ValueNotation ::= <empty> | <value STRING> | "("
ValueNotation ")" | ValueNotation "," ValueNotation
Instance ::= <empty> | "TAT" | "SUT"
Via ::= <empty> | <port STRING>
UCMSignal ::= Name UCMPParam
Name ::= <name STRING>
UCMPParam ::= <empty> | <param name STRING> |
UCMPParam "," UCMPParam
```

Fig. 5. Lowering rules language grammar

- It is prohibited to separate fields of variables values
- Only structures similar to SUT interfaces can be created
- Only constant template values and values that were obtained during concretization step are allowed

Limitation was made by creating a special language that is used to define lowering rules. Despite having all these limitations, user can define complex signal and protocol structure dependent on UCM signal parameters in accordance with language grammar. On Fig. 5, you can see the grammar in Backus–Naur Form.

##### B. User perspective

Custom editor called Lowering Editor was implemented to restrict user from making incorrect structures. Core features of the editor:

- Signal structure is taken from description of SUT interfaces
- Automated correctness check. It is impossible to save the rule with incorrect signal structure
- User-friendly way of displaying signals. Nested structures are tab-spaced
- Content assist for signal parameters and variables
- Basic text editing operations like copy, paste, undo and redo

Speaking of work process with the tool, at first all the signals in UCM are gathered in one place. After that user can select needed signal and edit its lowering rules in the editor (Fig. 5).

For selected UCM-level signal user can define lowering rules. As you can see on Fig. 6, rule consists of trigger condition and content. Content can be either one detailed signal, several signals or actions performed on the variables.

After specifying the condition and choosing the type of content, user can edit it in the right part of the editor. This part dynamically changes depending on what is selected in the middle of the editor.

For example, some signal was selected. Signals editor will appear in the right part of Lowering Editor (Fig. 7).

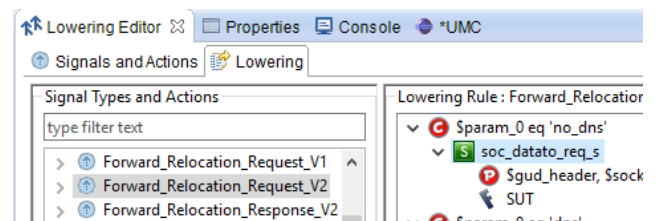


Fig. 6. Lowering editor with signal "Forward\_Relocation\_Request\_V2" being selected

Fig. 7. Signals editor

User selects the needed SUT interface in the drop-down list named "Select TDL Type or Template". Then user names the signal and puts concrete values in the fields of detailed signal.

Often similar conversion rules are required for different signals. Templates can be used to simplify this approach. A developer can define a template of detailed signal, specify either formula or concrete values as a parameter of detailed signal and then apply this template for all required signals. For each case of template usage a developer can specify missed values in the template, change the template itself or modify its structure without violating specified limitations. Templates mechanism simplifies significantly the process of conversion rules creation.

Consider the process of templates usage. Templates are created in separate editor (Templates Editor). In Fig.8 the template "template\_0" is shown which contains detailed data structures inside and the dummy values which shall be changed to concrete values when template is used.

Fig. 8. Templates editor

Note that template can be created only from SUT interfaces description or another template.

When a template of data structure is ready, it can be used for creation of conversion rules. Fig.9 represents usage of the template "template\_0" with substituted concrete values of signal parameters instead of the dummy value "value\_temp", which then will appear in behavioral MSC scenario.

Fig. 9. Template used in signals editor



Editor

---

Variable name:

Variable type:

Select TDL Type or Template:

Edit signal parameters in text below:

```
(.
192,
19,
2,
2,
32,
4,
3456
.)
```

Fig. 10. Contents of the variable "\$gud\_header"

In both signal and template editors user can use variables – some values that are too big to remember or retype every time. On the Fig. 7 all the values are taken from variables. Variables can be selected in the middle of the lowering editor. There are different types of variables with different editors and checks. For example, the contents of variable "\$gud\_header" used in "soc\_datato\_req\_s" detailed signal are shown on Fig. 10.

Variables can contain very complex structures and therefore greatly reduce expenses on creating detailed signals.

Overall process of selecting UCM-level signal, creating lowering rules and editing the resulting signal repeats for all UCM-level signals in the project

### C. Scenarios processing

Implemented module of behavioral scenarios conversion takes as an input the concrete behavioral scenarios and specified rules of conversion and the output is behavioral scenarios of the real system level, which can be used for testing. Overall scheme of conversion is shown in Fig.11.

Detailing stage is based on the grammar of data structures conversion rules described in Fig. 5 and conversion algorithm. The specific feature of test automatic scenarios detailing to the level of real system is allow to storing of proved properties of the system obtained in process of abstract model verification.

Based on the specified conversion rule each abstract signal in concrete behavioral scenario is processed. Signal parameters are matched to rule conditions and if the signal satisfies them, then it is converted into detailed form. Fig.12 shows concrete scenario, which will be processed.

In this scenario you can see 3 agents: "GTP#gtp", "GMG#gmg" and "GUD#gud". For example, we want to test an agent "GTP#gtp". On following trace it will be described as SUT.

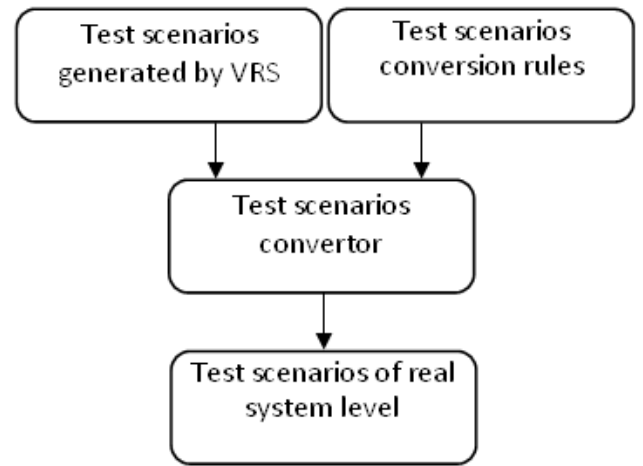


Fig. 11. Test scenarios conversion scheme

Other agents (or whichever we choose in the settings of the trace preprocessing) are marked as TAT and joined together.

After data structures conversion, concrete signals are replaced with detailed signals specified in lowering rules. Once simple signal structure unfolds in very complex nested data while still maintaining its correctness. You can see the results on Fig. 13.

## V. CONCLUSION

Proposed approach to behavioral scenarios generation based on formal models differs from existing approaches in using the process of automatic conversion of behavioral scenarios with abstract data structures into behavioral scenarios with detailed data structures used in real applications. Proposed language and overall scheme of this process allow automating of creation a set of covering behavioral scenarios.



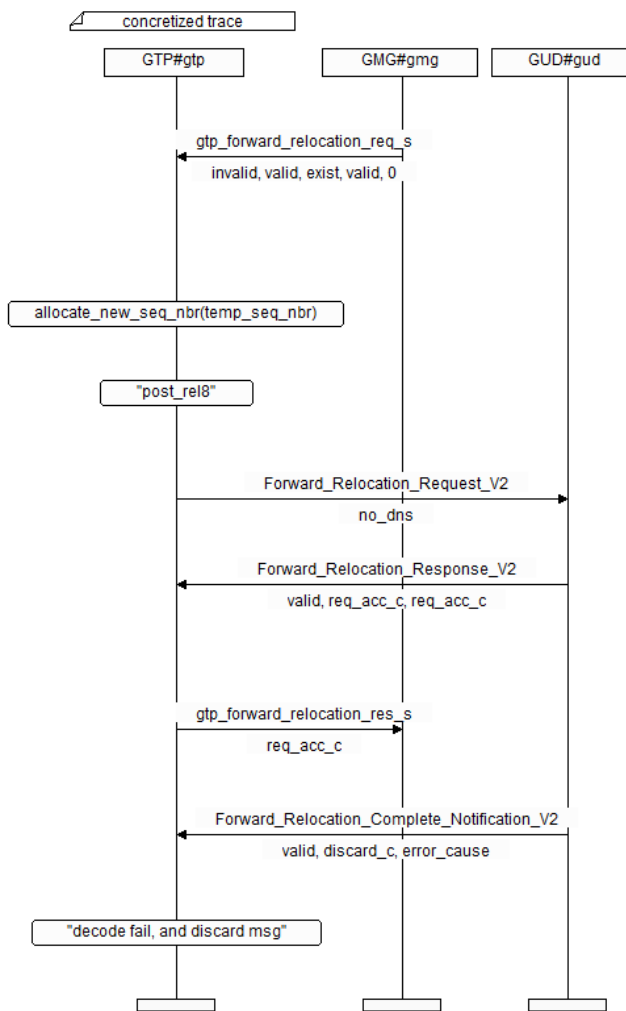


Fig. 12. Concrete scenario to be lowered

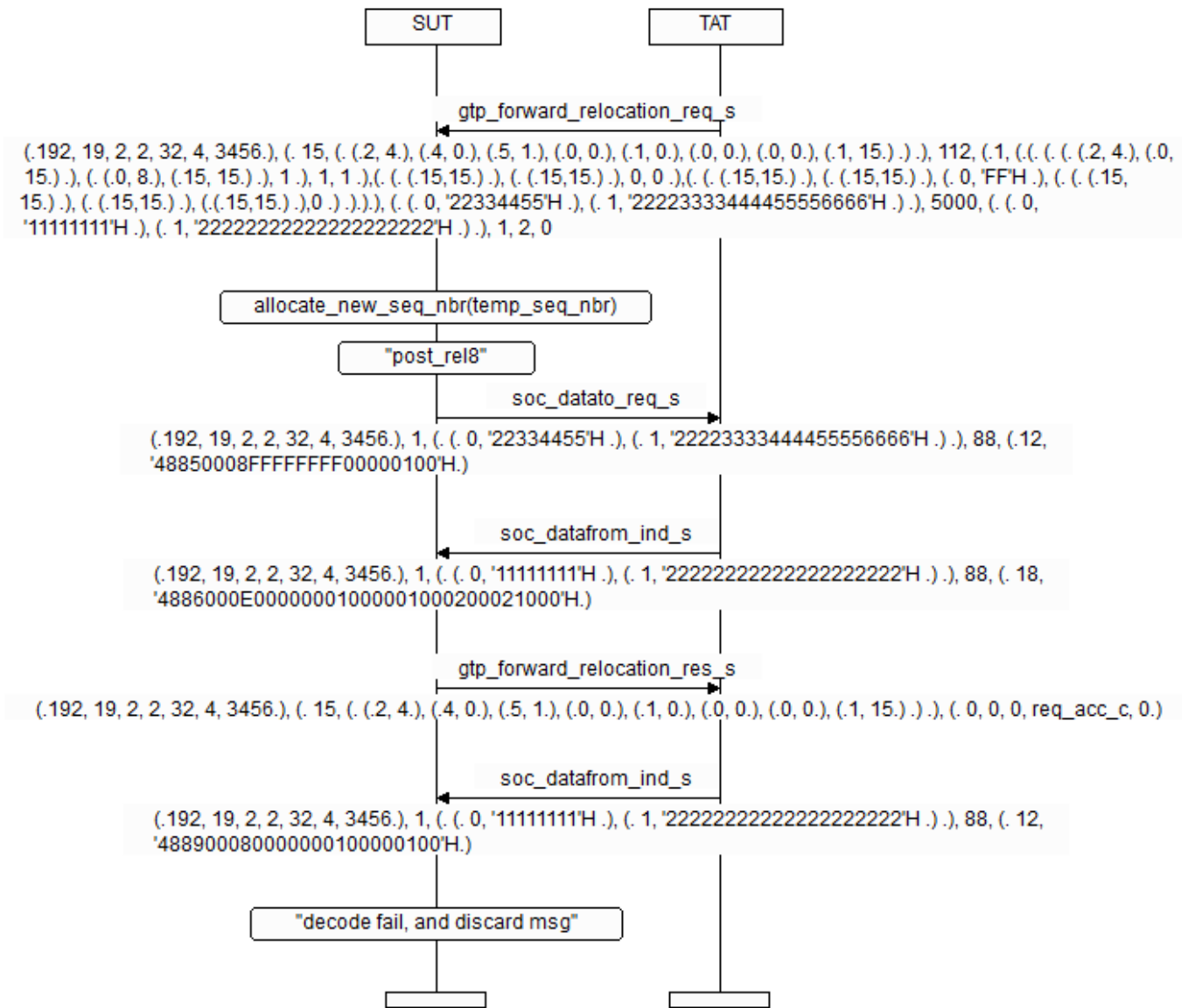


Figure 13. Lowered trace with detailed signals

In the scope of this work, the analyzer/editor for conversion rules of signals from abstract UCM model level into signals of real system level was developed and called Lowering Editor. It supports the following functionality: automatic binding between conversion rule and signal of UCM level, conversion rules correctness checking, templates usage, highlighting the syntax of conversion rules applying conditions specification, variables usage, libraries and external scripts (includes) usage, splitting UCM signal or action into several signals of real system in according to communication protocol, copy/paste/remove operations, import and export from/to storage file. Availability of described in the article features is able to make process of automatic conversion powerful and flexible for a different types of telecommunication applications.

Adding Lowering Editor into technology process of telecommunication software applications test automation allowed to exclude effort-consuming manual work in the cycle

of test suite automated generation for industrial telecommunication applications, increase productivity of test generation in 25% and spread the properties proved on abstract models into generated code of executable test sets. Excluding of manual work allow to reduce human factor in testing process and guaranty quality of generated test suite based on verification results.

## References

- [1] Model Driven Architecture- MDA (2007), <http://www.omg.org/mda>
- [2] Oscar Pastor, Sergio España, José Ignacio Panach, Nathalie Aquino.: Model-Driven Development. Informatik Spektrum, Volume 31, Number 5, pp. 394-407 (2008)
- [3] Sami Beydeda , Matthias Book, Volker Gruhn.: Model Driven Software Development.: Springer-Verlag Berlin Heidelberg, 464 p. (2005)
- [4] Robert V. Binder, Anne Kramer, Bruno Legeard, 2014 Model-based Testing User Survey: Results, 2014 <http://model-based->

testing.info/wordpress/wp-content/uploads/2014\_MBT\_User\_Survey\_Results.pdf

- [5] Buhr R. J. A., Casselman R. S.: Use Case Maps for Object-Oriented Systems, Prentice Hall. 302 p. (1995)
- [6] A.A. Letichevsky, J.V. Kapitonova, V.P. Kotlyarov, A.A. Letichevsky Jr., N.S.Nikitchenko, V.A. Volkov, and T.Weigert.: Insertion modeling in distributed system design // Problems of programming, pp. 13–39. (2008), ISSN 1727-4907.
- [7] I.Anureev, S.Baranov, D.Beloglazov, E.Bodin, P.Drobintsev, A.Kolchin, V. Kotlyarov, A. Letichevsky, A. Letichevsky Jr., V.Nepomniaschy, I.Nikiforov, S. Potienko, L.Pryima, B.Tyutin.: Tools for supporting integrated technology of analysis and verification of specifications for telecommunication applications SPIIRAN works №1-28p. (2013)
- [8] A. Kolchin, A. Letichevsky, V. Peschanenko, P. Drobintsev, V. Kotlyarov.: Approach to creating concretized test scenarios within test automation technology for industrial software projects. Automatic Control and Computer Sciences, Allerton Press, Inc., vol. 47, #7, pp. 433–442. (2013)

# Automation of Failure Mode, Effects and Criticality analysis

Peter Privalov

ISP RAS

25 Alexander Solzhenitsyn str.  
Moscow, 109004, Russian Federation

**Abstract**—Failure mode, effects and criticality analysis (FMECA) is one of the most powerful techniques to eliminate potential failures of products and assembling processes. This analysis can be performed at any stage of product design, thus lowering costs of errors. For a number of products this analysis is required by industrial standards, for others it can help to gain high quality. The purpose of this paper is to observe current implementations of automated FMECA and to make a summary of requirements to new implementations.

**Keywords**—FMECA; early validation of models; failure analysis;

## I. INTRODUCTION

To achieve success on market, product quality is of the great importance. Obviously, the production of the product full of defects or having a low service life will not bring success in the market. So manufacturers have to eliminate all potential failures from their products and assembling processes. There are many methods to troubleshoot, ranging from manual analysis. For industrial projects this approach is not applicable because of their scale. However, there are a number of formal analysis techniques that can reduce the probability of failures, optimize assembly processes and maintenance of a product. One of such methods is Failure Modes, Effects and Criticality Analysis (FMECA) [1], [2], [3], [4] that also often referred as FMEA. To avoid confusion, FMEA is a part of FMECA, but without quantitative assessment.

FMECA covered entire project's life cycle and potential failures which appear at each stage. This analysis examines components of the project under test to determine the possibility of failures in each component and their effort on the project. FMECA conducted repeatedly from the early stages of project development help to perform improvements in project at convenient state and decrease crisis caused by late changes. FMECA can reduce the chance or eliminate the necessity of corrective actions in the late stages of project development, that can lead to even greater problems than initial defects. Since the analysis is recommended to start at the earliest stages of a project design, it is applied to the model of the tested object.

The model is a description of the designed object, as a composition of components. At the beginning of the development process model is abstract and contains only the conceptual components of the future product, which clarifies during the design process. There are few ways to create the model. In that paper, we will describe models created with Architecture Analyze and Design Language(AADL)[5] and created graphically.

In FMECA process system's components are examined if there are any potential failures. For all that failures then studied

their severity rating, probability of occurrence and chance to detect and then calculates the risk priority number(RPN) that is usually multiplication of three previous ratings.

FMECA also examines how potential failure in the component may affect the system. So in the initial state all components are in working states and they may turn into a failed state because of inner failure or failure in the connected component. In FMECA one has to trace the propagation of failed states to understand the effect of failure.

In a number of industries, FMECA required by production standards. For example, ARP4761 standard for aviation or MIL-STD-882D for Department of Defense projects. These standards describe system safety and reliability and assessment methods. FMECA itself is a set of activities designed to identify and assist in eliminating of potential failures. In Russia conducting of FMECA regulated by GOST 27.310-95 described in [6]. Its summary is presented below.

At present, FMECA is commonly manual process that conducted in the final stages of industrial projects and required a team of highly qualified specialists. In this way, FMECA is an expensive and not very effective, as well as a singular analysis in the early stages of project development. But repeated analysis at each stage of the project, using data obtained in the previous stages can greatly improve the design. However, manual analysis is too expensive, and storage and use of obtained data in the preliminary stages are a non-trivial task in the development of an industrial project. Manual analysis of industrial objects required work of a team of several specialists. For some projects, there is a possibility to employ specialists for manual analysis, but there are a number of cases where such an analysis is required, but there is no possibility for manual analysis. This necessitates FMECA automation.

There are a number of software tools, partially automate the analysis. All them can be divided into two groups by the way they use to build the model. OSATE is one of tools that uses AADL and its annexes. The other group contains not AADL-based tools. From that group, the most interesting is RAM Commander. It was applied in many industries, that required FMECA by world-renowned companies. The purpose of this work is to analyze the comparison of these tools and to identify requirements to new implementations.

The rest of the paper is organized as follows. The second section describes FMECA standards and process. The third section contains a description of two automated tools and their comparison. The fourth section compares FMECA processes conducted by these tools. The fifth section concludes the paper.

## II. GOST 27.310-95 SUMMARY

Before work on the project begins, a plan of the analysis of each stage of work, containing recommended techniques, deadlines and control procedure must be drawn up. The analysis should begin from the earliest possible stages of the project's design and systematically repeated in later stages when appears new raw data for analysis (for example, adding information about the possibility of overheating of the component in the neighbourhood to the heat source). In each stage FMECA required checks for verification of the completeness of the implementation and effectiveness of the corrective actions recommended on previous stages. At any stage analysis begins with FMEA of the object. By the results of which a decision on the need for an in-depth quantitative analysis and evaluation of the criticality of certain types of failures is taken.

At each stage FMECA resulted in a report containing:

- formalized description of the object (model), indicating its level of specification to which (or from which) analysis was carried out;
- description of the analysis algorithm;
- completed worksheets that were used in the analysis;
- summary of the analysis, including: listing of possible failures classified by causes and conditions of occurrence, effects and criticality; lists of critical components and manufacturing processes;
- verdict about the possibility of the transition to the next stage of development of the object with required corrective actions, or suggestions for complete redesign of the project if the identified deficiencies cannot be remedied in the subsequent stages.

The level of specification of the object from which FMECA carried out at a certain stage of its development is set based on the required accuracy of result, technological and operational documentation; availability of the necessary raw data; degree of novelty of the object's design and its components and technologies of their manufacturing.

There are few FMECA types [7] that conducted for different purposes and in different design stages. For any analysis type the first phase of project development is a making up a list of critical components or processes. This list then systematically adjusted in each subsequent stage by eliminating elements for which the efficiency of improvements have been proved by analysis, calculations, experimental data, and including newly identified critical elements in the list. Then the list of critical elements approved by the customer.

Lists of the critical elements are to make up for each project. But there are some types of elements that commonly included in the list.

The list of critical elements includes:

- elements, which failures possible severity exceed the allowable level of the object;
- elements, which failure would inevitably cause a total failure of the object;

- elements with a limited lifetime that do not provide the desired durability of the object;
- elements, for which there are no reliable data on their quality and reliability at the time of analysis.

The list of critical processes includes processes, which affect on the quality and reliability of the object and its elements at the time of the analysis is unknown or poorly understood

## III. TOOLS OVERVIEW

Here will be described software tools in general and their features that absent in other tool.

### A. OSATE

OSATE described in [8] is an open-source tool platform to support AADL v2. In January 2012 correction to a number of errata to AADL v2 have been approved. These revisions, referred to as AADL v2.1, are supported by OSATE 2 and are summarized in [9]. This tool intends both end users and tool developers. For the former it provides a complete textual editor for AADL and a set of simple analysis tools. For the latter it provides a full support for the AADL meta-model on an Eclipse platform.

OSATE is a tool based on AADL that supports language annexes. Thus, AADL used to describe the system and its internal connections and its annexes are used, each for its own purpose. One of annexes – Error Model Annex (EMV) - just used for failure analysis like FMECA. In particular, it separates functional dependency from error propagation dependency. OSATE traces routes from all operating states to failed states and stops when system failure or recovery condition is detected. Checks conducted to prevent returns to the already considered state that prevents the appearance of endless loop, which is especially important for the recoverable system

One of the features of the AADL language is a separate definition of component type, its implementation and component instance. Also, there are a possibility of inheritance for component type and implementation. So properties declared for the type inherited by implementation and by child type if any, and the implementation itself can clarify properties. This mechanism is similar to the mechanism of inheritance of object-oriented programming languages. In the case of modeling language, it allows to develop model rapidly, and to avoid unnecessary duplication of code. For example, this allows one to describe the component once and to use it in many places. One can add local properties for FMECA to any implementation or even instance of that component and to change them during the model development.

### B. RAM Commander

RAM Commander as described by [10] is a comprehensive software tool for Reliability and Maintainability Analysis and Prediction, Spare Parts Optimization, FMEA/FMECA, Testability, Fault Tree Analysis, Event Tree Analysis and Safety Assessment. Its reliability and safety modules cover all widely known reliability standards and failure analysis approaches.

RAM Commander isn't an open source tool. It is a tool with a well-developed structure around it. There are a large number

of libraries of common components. This tool also allows to design a project in team through the network and import data from other engineering editors, such as CAD. RAM Commander is able to generate reports on FMEA \ FMECA, in a number of common standards.

Graphical editor of RAM Commander consists of two parts. The top part contains diagram of the developed product with product units or development stages and dependencies between them. In the lower part there is a table with information about potential failure modes (name, severity, frequency detection, recommendations) for the selected component. These properties are associated with the currently selected component instance. The editor also implements the classic for a graphics editor features such as drag-and-drop. If one uses it properties will be copied with the element.

An interesting feature of Ram Commander is the ability to analyze subsystems, declared in nested blocks. It helps to disassemble the system into functional or construction blocks, analyze them separately and then used the results of these analyzes to analyze entire system.

RAM Commander allows multiple users to edit the same project. First user has an exclusive access to the diagram (picture elements, their position, dependencies, etc.), and other users have access to FMECA data (analysis table at the bottom of the screen).

#### IV. COMPARISON

Both of the tools matches common FMECA standards, but not always in a usable way. So we will compare actions that one has to do at each step of FMECA and abilities provided by these tools.

1) Determination of the object. In OSATE model described by AADL language is the system under test. If the system is large then its description can be broken down into constituent parts. But to analyze any part separately it should not contain external dependencies. Similarly, in RAM Commander, all described model is tested object. For large scale models it can be more suitable because of the ability to perform FMECA for nested parts of the model, even if there are external dependencies. RAM Commander also can import libraries and lists of various characteristics of materials.

2) Types of analysis. In both cases, it is a bottom-up analysis. OSATE using the AADL model and RAM Commander using the project's parse tree.

Types of analysis:

- Design: in both systems, an analysis conducted in terms of described components.
- Process: both systems allow to describe manufacture processes and dependencies between them and between hardware, so process FMECA also can be conducted.

3) Determination of operating conditions. RAM Commander has the tool to define them. In OSATE that tool is absent, and the interaction of the system and environment described by the system designer within the terms that he considers suitable. For example, in terms of the events or failures.

4) Report building. RAM Commander supports many common methods of report generation, OSATE also supports some of them. Both systems allow user to customize the report.

5) Determination of the elements that contains potential failures. In OSATE an element properties are specified when the model developing. An interesting point is that the element can be marked as source of failures in the description of the type, in implementation or in instance of this element. In RAM Commander properties are specified for each element individually.

6) Identification of possible consequences: Error Model Annex of AADL language allows to describe the propagation of failures and show their effect on the overall system. This feature accurate the list of failures' effects. In RAM Commander it is possible to specify how the sub-component's failure will affect component as well as the dependence of the component from the other components on the same level. The difference in these approaches is that RAM Commander system model and model of failure effects propagation coincide, while OSATE allows the user to indicate the dependence between components in failure effects propagation independently from the system model.

7) Determination of the causes of failure and the probability of its occurrence. In automation FMECA this point, partly duplicates the previous action if the failure of a component caused by the propagation of failure from another component, or an event described in the system, otherwise the user must manually add information to the model. In that way this step realized in both systems.

8) Determination of methods of control and prevention failures. Usually, this process requires the inclusion of additional items in the model or instructions to staff. Both considered tools can't perform it automatically.

9) Calculation of the risk priority number (RPN), then user defines the required corrective actions, and then recalculates RPN with new obtained severity ratings, and the likelihood of occurrence of detection. In both systems one can determine how to calculate RPN and notations of ratings.

#### V. CONCLUSION

Considered software tools provide extensive functionality for the automation of FMECA, and many other types of analysis, such as fault tree analysis, the functional hazard assessment, dependency diagram. They offer two different approaches, each having its own pros and cons.

RAM Commander main advantage is the visibility of the developed model as well as the availability of infrastructure around the instrument. In OSATE - "object-oriented" approach to the description of the system, the ability to add new features to the tool, the cheapness of model refinement.

The main goal of automation is to enable carrying FMECA as often as possible in the development and production of the tested object without significantly increasing of the production cost. So the future implementations of automotive FMECA will meet the following requirements:

- The possibility of applying from the earliest stages, when there is only a general concept of the system

under examination to the stage of production of the final version of the product

- Development of a model should not be time consuming
- At each stage the model should be clear and vivid
- Ability to interact with a common engineering program
- Ability to generate reports in popular formats and in custom format
- Log history of the model changes
- Ability of joint work of the development team
- Formalized description of external factors and operating parameters

## REFERENCES

- [1] Ravindra Khare, Failure Mode & Effects Analysis: A Technique to Effectively Mitigate Risks.
- [2] Carl S. Carlson, Good FMEAs, Bad FMEAs - What's the Difference?, Reliability HotWire, July 2012, Issue 137
- [3] Kenneth Crow, Failure Modes and Effects Analysis, DRM Associates, 2002
- [4] Carl S. Carlson, FMEA success factors: An effective FMEA process
- [5] <http://www.aadl.info/aadl/currentsite/>
- [6] [http://www.snip-info.ru/Gost\\_\\_27\\_310-95.htm](http://www.snip-info.ru/Gost__27_310-95.htm)
- [7] [https://en.wikipedia.org/wiki/Failure\\_mode\\_and\\_effects\\_analysis#Basic\\_terms](https://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis#Basic_terms)
- [8] [https://wiki.sei.cmu.edu/aadl/index.php/Osate\\_2](https://wiki.sei.cmu.edu/aadl/index.php/Osate_2)
- [9] [https://wiki.sei.cmu.edu/aadl/index.php/Standardization#SAE\\_AADL\\_A\\_S5506B](https://wiki.sei.cmu.edu/aadl/index.php/Standardization#SAE_AADL_A_S5506B)
- [10] <http://aldservice.com/reliability-products/rams-software.html>



# Parallel processing and visualization for results of molecular simulations problems

Dmitry Puzyrkov\*, Podryga Victoria<sup>†</sup>, Polyakov Sergei<sup>‡</sup>  
Keldysh Institute of Applied Mathematics (Russian Academy of Sciences)  
Miusskaya sq., 4  
Moscow, 125047, Russia  
Email: \*dpuzyrkov@gmail.com, <sup>†</sup>pvtictoria@list.ru, <sup>‡</sup>polyakov@imamod.ru

**Abstract**—This paper presents "mmdlab" library for the interpreted programming language Python. This library allows to carry out reading, processing and visualization of the results of numerical calculations in the tasks of molecular simulation. Considering the large volume of data obtained from such simulations, there is a need in parallel realization of algorithms for processing those volumes. During the development process we have study the effectiveness of the Python language for such tasks, and we have examined the tools for it's acceleration and cluster computations. Also we have investigated the problems of receiving and processing the data, located on multiple computational nodes. As a tool for scientific visualization was chosen an open-source "Mayavi2" package. The developed "mmdlab" library was used in the analysis of the results of MD simulation of the gas and metal plate interaction. As a result we managed to observe the effect of adsorption in details, which is important for many practical applications.

**Keywords**—*parallel processing; visualization; molecular dynamics; Python; Mayavi2*

## I. INTRODUCTION

Advances in computer technology and the rapid growth of computational capabilities significantly increased the possibilities of computational experiment (CE). In particular, nowadays it is already possible to study the properties and processes in complex systems on molecular and atomic levels, for example, using molecular dynamics (MD) approach. Mathematical models, which describe such processes, may consider huge amounts of particles: up to billions of them, and even more. In addition, each particle can be described by dozens of parameters and the volume of output data in such CE can be estimated in terabytes. Processing of such volumes of data in serial mode can potentially take years, and optimization of computing code does not bring a significant acceleration of the computations.

Therefore, currently the most widely used approach to accelerate the large-scale computing is it's paralleling, which means that a great number of compute nodes would process a large amount of data each handling apart of it. As a result of paralleling, each node receives only a small part of the data set which is easy to manipulate with. This technique significantly reduces the time required to complete data processing, but leads to several problems concerning the data storage. Most often, after performing calculations compute nodes exchange the results of computations, and master process assembles them in RAM or in a storage device as one large array or a file. However, in the large-scale computations the size of the

result array (file) can significantly exceed the resources of the master node. In this case, each compute node stores the results in isolation. The last described method of storage has several advantages. The first one is the lack of need to sequentially read all the results for further processing (for example, for visualization purpose) because each computational node only reads it's part of the data. The second advantage is that each individual data file is typically not very large (compared to the full data set), and thus it takes less processing time. Such data can be reached in various ways, for example using a distributed file system, on-the-node-process reading, or using the applications allowing to send data over the network, such as the SFTP.

The scientific programs that store data in the form described above, are considered in this article. The results of the simulation based on the algorithm, described in the article [1] were used as a data set for studying parallelization capabilities of the developed "mmdlab" library.

One of the ways of CE data representation is a two- and three-dimensional visualization. In order to assemble a complete state of the simulation results, it is required to read and process the data from each compute node, which in itself is a resource-intensive task. In most cases, the calculated data formats and storage methods differ depending on the calculation program. Therefore, such programs usually have their own visualizer, and calculate all the necessary visualization data in the process of computation, collecting them on the master node. In this case the visualization is provided by the means of such programs (LAMMPS, and others). Another way is to save data in the well-known standardized containers (HDF5, VTK, and other), which are supported by the majority of software for scientific visualization. The problems of such methods of storage and rendering are the limited possibilities of the used visualization software in regards to visualization and post-processing, and in the case of well-known standards of data storage there occurs the problem of loading large files.

This paper presents an attempt to create a flexible tool that allows importing, processing and visualization of data from different sources, regardless of it's structure: whether the data is in known formats or distributed calculation results in a custom format. The results obtained using the computer program described in the article [1] were considered as a test case. In view of the parallel algorithms and storage features, this data can be a one big file that describes the general state of the simulated system, as well as a distributed data, processed by every computational process separately.

The results obtained from the simulation are the information about the interactions of the gas molecules with the metal atoms near the surface. This process is characteristic for many technological microsystems used in nanotechnology.

## II. PROBLEM STATEMENT

The problem of collecting and processing the distributed data obtained as a result of some calculation program has several key features. Firstly, it is the specifics of the problem domain. As a result of searching among the various simulation packages there has not been found suitable means for parallel loading of distributed data relating to the considered task. This problem drove us to do this research. Secondly, the scale of the input data can differ greatly. It can be a small one-dimensional array or a large number of files distributed across the various computational nodes and file systems. Such problems are usually solved by means of a software system that generated this data, or by development of a specialized "loader" tool, which understands input-output formats used by the calculation program. Thirdly, there is a need to process such results for convenient representation on charts or in 3D visualization. Due to the features described above, in this work we made an attempt to create a framework for the software complex with the following features:

- Parallel reading of data from different sources
- User-defined data formats support
- Custom data filters and processors support
- Data visualization solution

It is important to emphasize that in the case of development of such library its expandability has a significant role. It should be relatively easy to use the developed framework for processing the data stored in any format, and to integrate it with the other known solutions for visualization and data processing. As the initial stage of development we chose the problem of post-processing and visualization of the results obtained in work described in the article [1].

This task involves the consideration of all the listed features of the selected application, because of the distributed structure of the data in different computer systems with remote access to it via SSH.

## III. DEVELOPMENT TOOLS

There are many known solutions for task-based parallelizing and data visualization. Feature of these solutions is the difficulty of their use, setup and installation.

Among the known solutions for clustering can be noted Apache Hadoop. This is a large and complex solution, which implements MapReduce model for task-based parallel processing. However, for the considered problem, it has many unnecessary features, such as a distributed file system (HDFS) and requirement of installation on computational nodes.

For general scientific visualization there are a variety of software packages, for example, Paraview, VMD, Tecplot. Each of these software packages has its own format of data storage, and is also able to read the standardized formats. However, in the case of a custom data format or a complex

data distribution all of these solutions require implementation of a special data loader.

Taking all the above into account, we decided to add into the developed library the support of the integration into such packages, and its own visualization and clustering tools. Furthermore, "mmdlab" library has a minimum set of dependency and does not require installation on the compute nodes. In view of the need for the above-mentioned integration into well-known solutions, as well as the requirements posed by the expandability of developed framework, we decided to use an interpreted programming language Python, due to the fact that almost all of that packages use Python in their plug-in systems.

### A. Python

Python [2] is a widely used in scientific community general-purpose high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than it would be possible in languages such as C++ or Java. The syntax of kernel of Python is very simple and short, at the same time a standard library gives the large volume of useful functions and convenient data structures. It is also a cross-platform, so you can use it (with some restrictions), both under the MS Windows and Linux operating systems.

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management, full introspection, exceptions and multiprocessing. The developers community created a lot of computer science libraries, that makes Python one of the most commonly used languages for big data analysis and scientific calculations.

Though Python already has version 3, in this study we used Python version 2.7, in view of the fact that some used libraries (for example, Mayavi2) were written in Python 2.7, and Python 3 and Python 2.7 in some cases do not have backward compatibility.

### B. IPython

IPython [3] is an interactive shell for Python language, which adds an expanded introspection, additional command syntax, code highlighting and autocomplition. The main feature of this project is that it provides the core for Jupyter web-application, which allows to write scripts in Python, R, and BASH directly in the browser, as well as interact with the objects of visualization. In this work IPython notebook application has been selected as the web-control system.

## IV. ACCELERATORS OF COMPUTATIONS

Despite all the advantages of the main realization of the interpreter CPython, it is necessary to remember that the Python is a high-level interpreted programming language. It cannot provide high performance itself, due to the memory management system and dynamic typification. It is very easy to use, but if performance is critical it is necessary to implement CPU-critical code in C or C++, to avoid the overhead of interpreter calls. However, there are several technologies allowing to evade the low-level programming.

Listing 1. Numba and Numpy array multiplication

```

from numba import jit
@jit(nogil=True, nopython=True)
def numpy_numba_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider

def numpy_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider

```

Another big disadvantage of the CPython interpreter is associated with the speed and performance in multithreading. The last is caused by use of the GIL (Global Interpreter Lock) mechanism representing mutex (the elementary binary semaphore) which is not allowing different threads to process the same bytecode at the same time. Unfortunately, this lock is necessary, since the memory management system in CPython is not thread-safe.

The following methods were considered to avoid this limitations.

#### A. Numpy

Numpy [4] is an open source library for Python. It implements fast multi-dimensional arrays and plenty of parallel (vectorized) algorithms for linear algebra, Fourier transform and other applications. Since Numpy is written in C, the executable code of the library is compiled into native code, and there is no need for its interpretation, gaining significant speedups of the array-processing methods. The threads that run inside Numpy do not depend on the GIL, present in the CPython, and therefore its use accelerates the execution of algorithms by parallelization. Besides Numpy has detailed documentation that facilitates the development and maintenance of the software. All these features make Numpy reasonable choice for array processing in Python.

#### B. Numba

Numba [5] is optimizing Just-In-Time (JIT) compiler, which allows to accelerate the time-critical code by compiling it into native code. Unlike Cython, Numba does not require explicit type annotations (but supports it) and does not translates the code in C language, which simplifies the use of this technology. In order to show Numba which methods are needed to be optimized, the user must use the simplest means of Python language, called a decorator. Marked by the special decorator methods Numba optimizes and compiles to machine code using LLVM (Low Level Virtual Machine) infrastructure. With the ability to turn off the GIL, as well as the compilation to native code without using the Python C API (for the methods that operates elementary types), Numba compiler can generate more efficient and optimized bytecode. Numba also automatically vectorizes all that it can handle, utilizing the capabilities of multiprocessor systems to the maximum.

Table I compares the speed of execution of the same Python code (multiplication arrays with multiplying and dividing by a constant, see Listing 1), in one case without Numba, in the other using this technology. Testing was performed on a system with the Intel Core I7-3630QM CPU.

It should be noted that the algorithm shown in Listing 1 is not parallel in the means of code, and the vectorization is

TABLE I. NUMBA AND NUMPY PERFORMANCE COMPARISON

N	Numpy	Numba	Speedup
10 <sup>6</sup>	0.19 ms	0.07 ms	2.77
10 <sup>7</sup>	1.62 ms	0.74 ms	2.19
10 <sup>8</sup>	16.06 ms	7.4 ms	2.17

performed by Numpy. The Table I shows that Numba allows to speed up the execution nearly twice due to JIT compilation, without any special optimization, such as, most likely, would be needed while using any other tools, such as Cython.

## V. PARALLELIZATION TOOLS

Considering a GIL mechanism, presenting in CPython, the use of standard Python threads is not an effective solution for parallel processing. GIL does not allow multiple threads run simultaneously on different cores (within one interpreter process) even on a multiprocessor system. However, running multiple processes of interpreters, which can exchange data, completely solves this problem. The only distinctive in this case is that the launch of the process is a much more prolonged operation than starting threads, and usage of multi-process application on small data is not rational. There are several tools for easy management of such tasks.

#### A. Multiprocessing

Multiprocessing [2] is a standard library module that provides an interface to create and manage multiple interpreters processes. Its API is similar to the threading module of the standard library. It also adds some new features, such as the Pool class, representing the abstraction and control mechanism for a set of parallel interpreter processes. Multiprocessing also implements interprocess primitives, such as queue and mutex. It is also worth noting that each process of the interpreter works in separate memory space, therefore there is no need to worry about race conditions when writing or reading variables, unless they are declared as an object in shared memory. Communication between the processes of the interpreter within a given library is through interprocess communication channel, based on pipes, using the pickle module, allowing to "serialize" and "deserialize" the Python objects (serialization - the process of transferring any data structure into a bit sequence; deserialization - the restoration of the initial state of the data structure from a bit sequence). All the tasks of synchronization and object transferring are carried out by the Multiprocessing module. Therefore, the user does not need to solve the problem of confirming that all data used in the calculation has been updated.

Listing 2. Parallel Python and multiprocessing usage for multiple arrays summation

```
import pp
import numpy as np
ppservers = ("10.0.0.1", "10.0.0.2", "10.0.0.3", "10.0.0.4")
serv = pp.Server(ncpus = 2, ppservers=ppservers)
def mpsum(array):
    pool = multiprocessing.Pool(2)
    half = len(array)/2
    s = sum(pool.map(sum, [array[:half], array[half:]]))
    return s
arrays = [np.ones(5000) for i in xrange(10)]
imports = ("multiprocessing",)
deffuncs = tuple()
jobs = [serv.submit(mpsum, (a,), deffuncs, imports) for a in arrays]
s = sum([job() for job in jobs])
print s
```

### B. ParallelPython

ParallelPython (PP) [6] is a library used to solve the problem of clustering applications. Its implementation has a client-server structure and it requires installation of the server part on the compute nodes. However, the server program of the PP is a simple one-file script, that can be transferred into the node in any possible way. Because of the simplicity of PP interface, it allows to run a computational task on a parallel cluster in few lines of code. This library has its own load balancer, and it also monitors the status of nodes and redistributes tasks in case of non availability of one of them. With Multiprocessing module, ParallelPython allows simply and conveniently use all of the capabilities of the cluster computing. Listing 2 shows an example of summing up the plurality of arrays in parallel mode, using ParallelPython and Multiprocessing. At every computational node two processes start by ParallelPython and each of them starts other two process by means of Multiprocessing. It is worth noting that this library, as well as Multiprocessing, uses the "pickle" module to serialize data and tcp / ip network messaging.

## VI. VISUALIZATION TOOLS

As it was already mentioned, there are many third-party tools for data visualization. The "mmdlab" library presented in this work can be used as a tool for preparation of data for the visualization in such packages, however it was also decided to add its own visualization capabilities. During the research it has appeared that the listed below libraries almost do not concede in options to the well-known packages for scientific visualization.

### A. Mayavi2

Mayavi2 [7] is a Python framework, which allows to build a general-purpose scientific visualization. It gives user a possibility to load and render the data in a separate GUI application and also has a convenient Python API for scene construction and rendering. This library is built over the well-known in scientific community VTK library. Mayavi2 gives ample opportunities for the visualization of data, beginning from hydrodynamic calculations and finishing with atomistic data. In the case of the interactive GUI mode, tools for changing the rendering parameters, such as the size of objects, color schemes, filter settings are also available. Mayavi2 also has a possibility of the offscreen-rendering (without displaying

image), that is extremely important for the server, distributed and batch operation of a large number of data. Listing 3 and the Fig. 1 show an example of the density distribution calculation of points and its three-dimensional visualization using Mayavi2 and library for scientific computing SciPy.

### B. Matplotlib

Matplotlib [8] is a Python library for building high-quality two-dimensional graphs. It is widely used in the scientific community. Usage of Matplotlib is very similar to the usage of the plot methods in MATLAB, however, they are independent projects. It is particularly convenient that the plots, which are drawn with the help of this library can be easily integrated into applications written with different libraries for GUI construction. Matplotlib can be integrated into applications written using the wxPython, PyQt and PyGTK libraries. Matplotlib module is not included in the standard library, but it is the de facto standard for the visualization of numerical information.

## VII. DISTRIBUTED DATA ACCESS

The data obtained from the algorithm, described in the article [1] has distributed structure, and is stored on the compute nodes, used for simulation. Fig. 2 shows an example of such data arrangement.

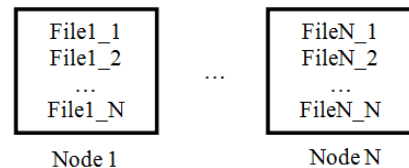


Fig. 2. Data distribution structure.

The composition of all the files is a complete form of the system simulated by means of molecular dynamics. It happens that the computational nodes use the shared disk space, for example, by means of the NFS (Network File System). However, access to the data from the client-side which needs to read and process the data is open only via SSH. Paramiko library can be used to solve this problem.

Listing 3. Kernel Density Estimation calculation and visualization script using SciPy and Mayavi

```
import numpy as np
from scipy import stats
from mayavi import mlab
mu, sigma = 0, 0.5
x,y,z = [10*np.random.normal(mu, sigma, 100) for i in [1,2,3]]
kde = stats.gaussian_kde(np.vstack([x,y,z]))
xmin, ymin, zmin = x.min(), y.min(), z.min()
xmax, ymax, zmax = x.max(), y.max(), z.max()
xi, yi, zi = np.mgrid[xmin:xmax:30j, ymin:ymax:30j, zmin:zmax:30j]
coords = np.vstack([item.ravel() for item in [xi, yi, zi]])
density = kde(coords).reshape(xi.shape)
figure = mlab.figure('DensityPlot')
grid = mlab.pipeline.scalar_field(xi, yi, zi, density)
min, max = density.min(), density.max()
mlab.pipeline.volume(grid, vmin=min, vmax=max + .5*(max-min))
mlab.points3d(x, y, z, scale_factor=1)
mlab.axes()
mlab.show()
```

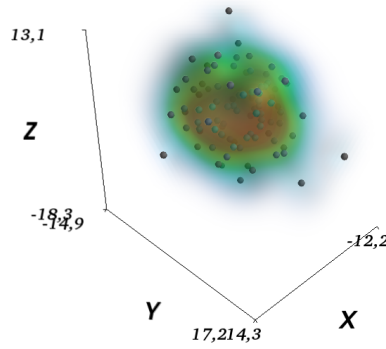


Fig. 1. Listing 3 execution result: Kernel Density Estimation as volume visualization

#### A. Paramiko

Paramiko [9] is a library for the Python language, which provides implementation and interface for interacting with remote systems via SSHv2 protocol. This library has both client and server implementations. In addition, Paramiko provides a convenient API, which implements objects of "file" type, which are representing files on the remote filesystem. This functionality was used as a basis for the implementation of SSH collector in the represented work.

### VIII. IMPLEMENTATION DETAILS

Using the tools above, there was initiated the development of the software complex, allowing to achieve the objectives, namely the parallel data reading and processing, as well as their visualization. As an initial stage, "mmdlab" package was written which implements a general purpose API for such tasks. Below are described the implementation problems we have to handle, application and solutions with the means of the developed library. There is also drawn further attention to the implementation peculiarities in some parts of the package.

#### A. Parallel data access

A module for reading and partial processing of the input data was named "datareader". In this module have been implemented the necessary objects for reading and representation of the data, such as Container, Parser and means of access to the files on the local file system and via SSH. In the terminology of "mmdlab" package, Container is a structure that stores the read data in a user-defined format. Parser is a special object that reads binary data structure and parses them, thereby obtaining a container. The Parser class receives the raw data from the Transport object that provides an interface for the access to the local or remote file system. Inheriting and combining objects from these classes, the user can easily make the loader, that parse a custom data format, and accesses it using any protocol, such as SSH or HTTP. On the Fig. 3 are shown the "mmdlab" components interactions.

Let's consider the reading procedure of the MD system's particular state described the article [1]. Given the distributed structure of input data, a single state of the system is a set of files of the atomistic data. For each of them it is necessary to read, parse and compile binary structure into a single container that contains the representation of the simulated system.

For the performance needs it is necessary to use a parallel

Listing 4. A part of DistributedDataReader class

```
class DistributedDataReader:
    ...
    def read(self):
        ...
        files = self.transport.list(self.file_mask)
        container = self.container()
        pool = Pool(processes=self.np, maxtasksperchild=self.mtpc)
        results = [pool.apply_async( rd,
                                   args=(f,self.transport.filer(),self.parser))
                  for f in files]
        for ct in results:
            container.append_data(ct.get())
        return container.finalize()
```

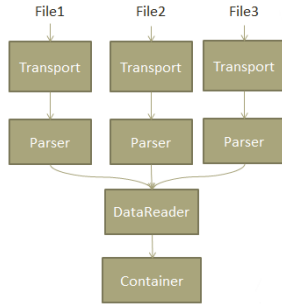


Fig. 3. MMDLAB components scheme.

algorithm for the reading and processing of the data. Master process launches  $N$  slave-processes that are able to load and parse the data. Then it begins to give every data file address to a every free process. When the slave process has finished the reading and parsing procedure, and assembled its part of the container, the master process combines the loaded data with its master container, and then assigns a new file to the slave process. After all the slave processes are completed, and there are no more files for reading, master process provides the necessary post-processing for the container, where all of the available data is stored, and sends it to the next data processor in line. It should be noted that in some cases it is not necessary to send all the data to the master host. For those cases, the "mmdlab" supports a possibility to use the post-processing pipeline in the slave processes, so they can make necessary calculations and send back only the result, but not all the processed data set. In order to enhance the ability of "mmdlab" package for reading the custom-format data, it is required to describe the new entity for storage and loading of such data. As an example, consider the implementation of such entities for reading a CSV (Comma-Separated Values) format with three columns. Listing 5 shows an example of such an extension to CSV reading from remote file systems via SSH.

In practice the user will need to describe the new class inherited from the class DummyContainer and to redefine the `append_data` method in it. Also it will be required to describe the class for raw data parsing.

### B. Pipeline

In this work, to run reading and processing tasks, it is proposed pipeline-type interface (see Listing 6, the `mmdlab.run` part). This method makes it possible to run an execution of a

Listing 5. CSV Container and Parser implementation using "mmdlab" package

```
package
class CsvCtr(dr.containers.DummyContainer):
    def __init__(self):
        self.cols = [[], [], []]
    def append_data(self, data):
        for i,d in enumerate(data[:]):
            self.cols[i].extend(d)
class CsvParser(dr.parsers.DummyParser):
    def data(self):
        cols = [[], [], []]
        for line in self.transport.readlines():
            c = line.split(",")
            for i in range(0,3):
                cols[i].append(c[i])
        return cols
nodes = \
({ "ip": "10.0.0.1", "pwd": "123", "login": "test",
  { "ip": "10.0.0.2", "pwd": "123", "login": "test" })
remotedirs = [(sys.argv[1], node) for node in nodes]
transport = dr.transport.RemoteDirs(remotedirs)
parser = CsvParser()
rdr = dr.DistributedDataReader(file_mask="1*.csv",
                              transport=transport,
                              parser = parser,
                              container = CsvCtr)

container = mmdlab.run([rdr, ])
```

chain of actions in one line, each of which is carried out over the result of the previous task. Also parallel operations over the same result of the previous method are supported.

For example, the call of `mmdlab.run([generate, [f1, f2, f3], sum])` first performs the "generate" method, then in parallel mode it runs three processes: "f1", "f2", "f3" each operating on the result of the "generate", and in the end it will summarize the obtained values. Restrictions on objects in the pipeline are simple: the object has to be callable, it should take the data for processing as an argument and it should return an object.

At the current stage of development, when you run a multithreaded processing over the previous action the result will be copied to each of the child process. In the future we plan to add some additional entities, allowing to manage the execution workflow, such as a special object that allows to perform an action in the master process, and to send the result's parts to the slave-processes. This may be necessary, for example, for the separation of the array into a multiple parts, and process each in a separate slave-process without sending the entire array to it.

Due to the fact that the pipeline is implemented by means



of the interface module Multiprocessing, consider some of the problems encountered.

### C. RAM leak in parallel processing

Let's consider the reading procedure of the DistributedDataReader class (see Listing 4).

During the testing it was found that a resources leak appears in the multiprocessing mode. After starting the pool of processes, and performing a variety of tasks in it, memory consumption increases dramatically. It became apparent that by default the started by Multiprocessing library interpreter processes handle all the scheduled tasks without restarting. Each task which is carried out in such processes leaves the context, which becomes bigger in the volumes of consumed memory as the more data the task returns. As a result, after long-term execution of multiple tasks at the computational node the RAM came to an end. The proposed solution of this problem is as follows. The object of a processes pool has a special parameter of the constructor named "maxtaskperchild", allowing to set the number of tasks that a single interpreter process can handle. When the counter of finished jobs becomes more then this value, the master-process algorithm will restart the interpreter.

Changing this parameter allows to vary the maximum amount of memory consumed. However, it should be noted that the smaller the value, the more often the master process will restart child processes' interpreters. It can take noticeable amount of time. Within the considered task of processing large amounts of data, the time is not critical, and installation of rather small value is quite justified because of memory limits.

Fig. 4 shows the dependence of the loading time on the "maxtaskperchild" parameter. The loader uses multiprocessing module, with the pool consisting of one process, and loads 256 data files in serial mode.

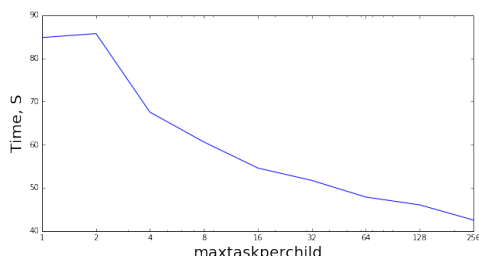


Fig. 4. Loading time of 256 files depending on a "maxtaskperchild" parameter, logarithmic scale

Taking into the account the Fig. 4, the optimal behavior of the processes pool is to restart the slave-workers every 16 tasks. It makes possible limiting the consumption of RAM and at the same time keeps the overhead of the interpreter restart time influence almost negligible.

### D. Multiprocessing and Pool of Pools

Another problem encountered in the development process is the fact that the default multiprocessing library does not allow to create "nested" pools for processes. In particular, if there appears a necessity to run in parallel the processes of reading

a plurality of states of the studied system (this will start new slave-processes that should start a lot of reading processes), for example, for the particles' trajectories construction, so the Multiprocessing module will not allow to do it.

The introspection which is supported by the Python language fully helps with the solution of this problem. The "mmdlab" package developed in this work has a construction shown in Listing. 7 included in it.

Listing 7. MultiPool class, allowing to run pool of processes inside child process, created by multiprocessing module

```
import multiprocessing
import multiprocessing.pool
class NoDaemonProcess(multiprocessing.Process):
    def _get_daemon(self):
        return False
    def _set_daemon(self, value):
        pass
    daemon = property(_get_daemon, _set_daemon)
class MultiPool(multiprocessing.pool.Pool):
    Process = NoDaemonProcess
```

It redefines the \_get\_daemon and \_set\_daemon methods at the "multiprocessing.Process" class and provides a new object, inherited from the Pool class. It should be used instead of the standard Pool class from Multiprocessing module.

### E. Data processing

For processing and filtering data in developed "mmdlab" library the same mechanisms as for the data reading are used. The so-called "pipeline" architecture is used which implicates the container object passing through a chain of a great number of data processors, that can change, supplement a container or create a new one. The "run" method in the "mmdlab" package passes the container obtained from the previous task to the input of the next processing method. The implementation of these processing methods can be both serial and parallel.

In the application to the analysis specific objective of molecular dynamics simulations' results from the article [1], the objects for data post-processing have been added to the developed library. For example, a filtration of particles by various criteria, in particular for getting the particles only from specified area, for filtration by indexes and division of particles according to physical materials. All computationally intensive procedures were optimized by using Numpy and Numba.

As a simple example, let's consider the task of visualizing of the particles' position and temperature that are divided by criteria of physical material in the predetermined area. Such problem can be solved using "mmdlab" library in the following way (see Listing 6). First, the user creates an object of the data loader, setting their location in the filesystem and a time mark. Then they need to specify the description of particles, which the division filter will work with, and create the corresponding objects of filters (the location filter and the division filter). Lastly they need to pass these objects to the pipeline. Calculation of temperature is performed during the container's post-processing stage. Listing 6 and Fig. 5 show the listing of such task and the execution results.



Listing 6. Reading, processing and visualization of the atomistic data using "mmdlab" package

```
from mmdlab.datareader.shortcuts import read_distr_gimm_data
import mmdlab
import sys
reader = read_distr_gimm_data(sys.argv[1],sys.argv[2])
filter_reg = mmdlab.dataprocessor.filters.RegionFilter([0,10,0,10,0,10])
parts_descr = \
{ "Nickel" : { "id" : 0, "atom_mass" : 97.474, "atom_d" : 0.248}, \
  "Nitrogen" : { "id" : 1, "atom_mass" : 46.517,"atom_d" : 0.296} }
filter_split = mmdlab.dataprocessor.filters.SplitFilter(parts_descr)
container = mmdlab.run([reader, filter_reg, filter_split ])
met,gas = container["Nickel"], container["Nitrogen"]
mp = mmdlab.vis.Points3d(met, scalar=met.t, size=met.d,
                        colormap="black-white")
gp = mmdlab.vis.Points3d(gas, scalar=gas.t, size=gas.d, colormap="cool")
mmdlab.vis.colorbar(gp, "Gas_T")
mmdlab.vis.show(distance=20)
```

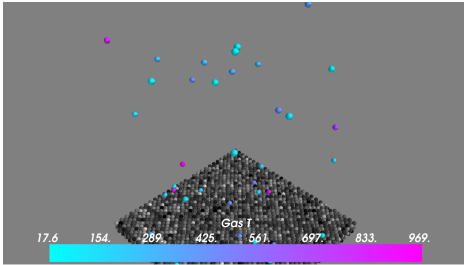


Fig. 5. The result image produced by execution of Listing 6.

### F. Cluster processing

For testing of the cluster mode was used the combination of the master node with the Intel Xeon E5-2650 (32 cores) and 6 compute nodes (Intel Xeon 5150 2.66 GHz, 24 cores) with shared file system over NFS. It gives certain freeness in respect of access to the data: it is not required to associate the input and the node on which processing is started, as any datafile is available from any of nodes. However, such configuration has a bottleneck: the storage input-output performance. As a result, it was decided to use the following strategy: a master node, which is a physical data storage, in the multiprocess mode loads data into memory and sends it to the cluster nodes in the form of internal representation, without the data processing.

In contrast to the strategy of "reading on each node" the described way allows to use the computational capabilities of the subordinated nodes on maximum, with the minimum input-output waiting, maximizing disk input-output utilization. In case of difficult visualization for which processing and rendering takes more time than reading one system state, such approach allows to reduce the average time of full processing almost to the data reading time, which is the potential minimum time of processing.

As an example of clustered task, consider the problem of constructing three-dimensional field of the gas density in the computational domain using Kernel Density Estimation (KDE) algorithm, implemented in SciPy library (see Listing 8). The graphs of execution time (see Fig. 6) and the acceleration (see Fig. 7) of such calculations, depending on the number of processors for a variable number of subtasks are shown below.

Listing 8. KDE Clustering example using mmdlab package

```
import sys
import mmdlab
from scipy.stats import *
from mmdlab import parallel
from mmdlab.datareader.shortcuts import *
from mmdlab.dataprocessor.filters import *
rdr = read_distr_gimm_data(sys.argv[1],0)

def calc_kde(kde, data):
    return kde(data.T)

parts_descr = { "Nickel" : { "id" : 0},
                "Nitrogen" : { "id" : 1}}
filter_split = SplitFilter(parts_descr)
filter_reg = RegionFilter([0, 100, 0, 100, 0, 100])
cont = mmdlab.run([rdr, filter_reg, filter_split])
gas = cont["Nitrogen"]
kde = gaussian_kde(np.vstack([gas.x,gas.y,gas.z]))
xi, yi, zi = np.mgrid[0:gas.x.max():30j,
                      0:gas.y.max():30j,
                      0:gas.z.max():30j]
c = np.vstack([item.ravel() for item in [xi,yi,zi]])
cores = sys.argv[2]
nodes = ("192.168.6.15","192.168.6.20")
cluster = parallel.Cluster(nodes)
args = [(kde,a) for a in np.array_split(c.T, cores)]
cluster.map(calc_kde, args)
density = np.concatenate(results).reshape(xi.shape)
```

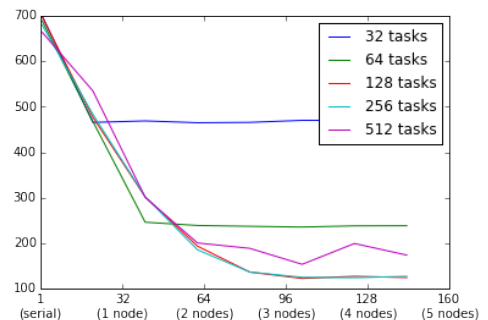


Fig. 6. Processing time for parallel KDE algorithm with various number of subtasks, depending on the number of used processors

It should be noted that if the number of tasks is less than the number of master node processes (which is up to 32), then the increasing of the process's count in this calculation is not

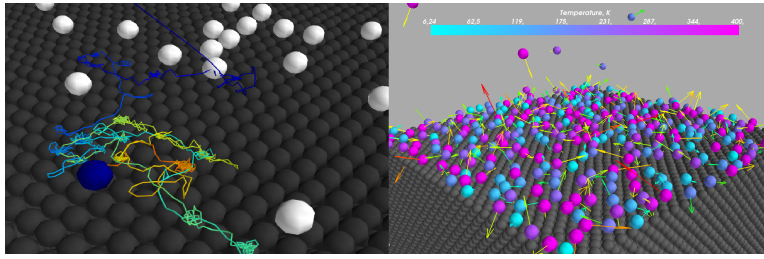


Fig. 8. Adsorption of Nitrogen on nickel plate and particle trajectory visualized using the "mmdlab" package.

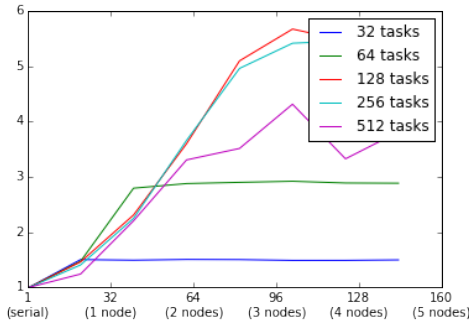


Fig. 7. Speedups for parallel KDE algorithm with various number of subtasks, depending on the number of used processors

effective. Also, the acceleration increases with the number of nodes involved in the computation, rather than with the number of actual processes. This is due to the following two features:

- PP considers that the overhead of process start-up and data transfer is significantly less on the master-node, than on the slave-nodes. Thus, it loads the master node to the maximum, before it starts to send jobs to the slave-nodes.
- Numpy already vectorizes array operations over all available cores, and the addition of a new processor will not make a significant acceleration.

Also we need to note that the PP, which is used as a library for clustering, automatically distributes the load across nodes, depending on the tasks execution time. So it makes sense to divide the original problem into a number of subtasks more than the number of available processes, if there are some "weak" nodes in the cluster. In this case PP forms a queue and gives tasks to the nodes taking into account efficiency of each node, thereby providing a load balancing.

### G. Visualization

For the visualization in this work Mayavi2 and Matplotlib library were used. For convenient usage of the common rendering methods, the "mmdlab.vis" module was included, which is a wrapper over the methods of these libraries, combining their capabilities to achieve the desired result. Due to the single-threaded architecture of Mayavi and Matplotlib, data visualization process is currently supported only in the single-threaded mode within a single process. However "mmdlab" allows to run a hybrid task of reading and rendering on a set of nodes and in the multiprocess mode, which significantly

accelerates the rendering of frame-by-frame video animations. For example, consider the task of rendering an animation which consists of frames representing the state of the studied system in consecutive timepoints. Basic data can be distributed across the multiple nodes, thus the visualization can be run on each of the nodes, and then the result can be collected on the master-node. The following algorithm is proposed for the solution of such a problem:

- On each of the specified nodes run a sequence of reading and visualization
- Collect all the frames that were drawn on the master node
- Assemble an animation from collected frames

To build an animated GIF format file "mmdlab" library uses the program "convert" from the ImageMagick [10] utils.

## IX. CONCLUSION

This paper presents the experimental version of a high-level library "mmdlab" for the Python language. Usage of such library makes it possible to perform a simple clustering and paralleling for the various types of processing tasks, such as reading, post-processing and visualization. It can operate over the large-scale data, distributed over the computational nodes in parallel mode. The main tasks of the development of this library are the analysis and visualization of the data obtained as the result of MD simulation of gas-metal microsystem described in the article [1]. To achieve this goals it was necessary to process about 1.3 TB of data obtained from one simulation, and there were three simulations with different materials temperatures. Usage of the "mmdlab" library allowed to closely observe the effect of nitrogen adsorption on a nickel plate (see Fig. 8) including an analysis of the individual particles' trajectories. Special attention was paid to a possibility of extension of the created library. It is possible thanks to flexibility of the used tools. As a result, usage of the developed library can be extended to reading and visualization of potentially any structures of data.

## ACKNOWLEDGMENT

Work is performed with assistance of the Russian Foundation for Basic Research (grants No. 15-07-06082-a, No. 15-29-07090-ofi\_m).

## REFERENCES

- [1] V.O. Podryga, S.V. Polyakov, D.V. Puzyrkov, Supercomputer Molecular Modeling of Thermodynamic Equilibrium in GasMetal Microsystems (in Russian), in Vychislitel'nye Metody i Programirovanie, vol. 16, no. 1, pp. 123-138, 2015.
- [2] Python official documentation [Online]. Available: <https://www.python.org/>
- [3] P. Fernando, E.G. Brian, IPython: A System for Interactive Scientific Computing (in English), in Computing in Science and Engineering, vol. 9, no. 3, pp. 2129, 2007. (2015, Feb. 4), [Online]. Available: <http://ipython.org>
- [4] Numpy official documentation [Online]. Available: <http://www.numpy.org/>
- [5] Numba official documentation [Online]. Available: <http://www.numba.pydata.org/>
- [6] ParallelPython official documentation [Online]. Available: <http://www.parallelpython.com/>
- [7] Mayavi2 official documentation [Online]. Available: <http://docs.entthought.com/mayavi/mayavi/mlab.html>
- [8] Matplotlib official documentation [Online]. Available: <http://matplotlib.org/>
- [9] Paramiko official documentation [Online]. Available: <http://www.paramiko.org/>
- [10] ImageMagick official documentation [Online]. Available: <http://www.imagemagick.org/>

# Memristor-based Hardware Neural Networks Modelling Review and Framework Concept

Dmitrii D. Kozhevnikov

Faculty of Computer Sciences  
National Research University Higher School of Economics  
Moscow, Russia  
ddkozhevnikov@edu.hse.ru

Nadezhda V. Krasilich

Faculty of Business Informatics  
National Research University Higher School of Economics  
Perm, Russia  
nadezhda.krasilich@mail.ru

*This paper is a report of study in progress that considers development of a framework for modelling hardware memristor-based neural networks. An extensive review of the domain has been performed and partly reported in this work. Based on this review, a number of development requirements is derived and formally specified, ontological and functional models are proposed to foster understanding of the corresponding field.*

**Keywords**—memristor; memristor model; hardware neural network model; memristor-based neural networks.

## I. INTRODUCTION

Until 1970-s the world has been aware of only three passive elements of electrical circuitry: resistors, capacitors and inductors. The three stated elements coupled with natural relationships provide five connections for four basic notions of electrical circuit theory (voltage, charge, current and flux). Mathematics, however, claims that four things can be mutually interconnected in six different ways. Indeed, the relation between charge and flux was not present. It wasn't until 1971 that the discordance has been formulated and solved. A new element – memristor - has been proposed by Leon Chua in his paper in IEEE Transactions on Circuit Theory completing the mathematical symmetry of circuit theory. It took nearly 40 years for memristor to transform from a purely theoretic concept into feasible implementation. In 2008 a group of scientists from Hewlett-Packard Labs lead by Stan Williams has finally built working memristors [1].

One of the most promising domains of memristor application, seem to be artificial neural networks [2]. These often come in either software or hardware implementations, sometimes in a combination of both. While digital neural networks simulate the data processing mechanism of biological neural networks, hardware ones strive to emulate it. It is worth mentioning that since most of computer architectures conform to the von Neumann architecture, neural network simulation becomes a challenging task because of the paradigm mismatch.. Instead of simulating the ways of nature, hardware neural networks try to directly replicate them, creating non-von-Neumann architectures. In comparison with digitally simulated networks, hardware ones can achieve better speed, less power consumption and chip space.

On the other hand, hardware networks often prove to be far less accurate than their software counterparts, due to the nonuniformity of analog components [3]. Another disadvantage

of modern hardware neural networks, which they actually share with the software ones, is the volatile storage of synaptic weights. There are ways to achieve the nonvolatile weight storage within hardware networks, but usually such weights are either static (cannot be changed once manufactured), quickly digress (require frequent updating) or are rather hard to program [4]. The emergence of memristor, however, seems to have opened new possibilities in addressing the stated problems. Memristors seem to be a perfect match for synapses, making hardware implementations of neural networks more reliable and greatly increasing productivity of neural computations [5].

Nevertheless, memristors are still scarcely available and lack industrial-grade production. Being such a new technology, they are often hard and expensive to acquire for experimentation, but a large variety of memristor models has already been produced, making it possible to model memristor-based devices.

Thus, considering the domain of artificial intelligence, a need in profound and correct model of artificial memristor-based feedforward neural network arises. Such model would be of great help in assessing the qualities of modeled system: computation performance, time and energy expenses, material costs, etc. Consequently, the goal of the research is to develop a framework for modelling artificial memristor-based neural networks.

## II. THEORETICAL MEMRISTOR

The concept of memristor has been recognized since 1971, when Leon Chua has proposed for the first time in a well-organized and mathematically described way [6].

The 1971 Chua's paper in IEEE Transactions on Circuit Theory is considered to be the pioneer work in the corresponding field of research. Although, the concept of memristor-like devices has been suggested earlier in 1960 by Bernard Widrow, Leon Chua was the first one not only to provide a feasible foundation for memristor's existence, but also to estimate and mathematically describe its' supposed behavior and properties.

Memristor fulfills the mathematical symmetry of relationships between major circuit notions. The relationship created by a memristor, according to Chua, is expressed as follows:

$$v(t) = M(q(t))i(t),$$

where  $M(q(t))$  is the memristance defined as

$$M(q) \equiv \frac{d\varphi(q)}{dq}.$$

The definition of memristance may be represented in a more convenient form by substituting flux and charge with their integral definitions:

$$M(q(t)) = \frac{d\varphi/dt}{dq/dt} = \frac{d[\int_{-\infty}^t v(\tau)d\tau]/dt}{d[\int_{-\infty}^t i(\tau)d\tau]/dt} = \frac{v(t)}{i(t)}.$$

The similarity of memristor to the remainder of classical circuit elements can be better reflected by expressing their definitions via differential equations as it is done in Table I.

The *first important property* of memristors, which commonly is referred to as **memristance** and stands for the ability to change its resistance gradually via a controlled mechanism (e.g. memory of device's history of charge).

The *second significant attribute* of memristors, figured out by Chua, is the **non-volatility** property, which stands for the absence of internal power supply. In other words, Chua proposed that memristor is able to store the value of own resistance without the need to be connected to a power source.

In 1976, Leon Chua and his fellow colleague Sung Kang proceeded exploring the mathematical and physical properties of the memristor [7].

They had come to an understanding, that since memristor is a dynamic device, one equation is not enough to describe it, henceforth memristor's behavior is represented by following equations for current-controlled memristor

$$x = f(x, i, t)$$

$$v = R(x, i, t)i$$

and for voltage-controlled one

$$x = f(x, v, t)$$

$$v = R(x, v, t)i$$

Where  $v$  and  $i$  denote the input voltage and current respectively and  $x$  stands for the internal state of the device. In their paper, Chua and Kang also provided a more generalized concept of memristive systems with no specific reference to particular physical variables.

One noteworthy peculiarity derived from these equations is that regardless of the state  $x$  (which implements the memory effect), the output voltage is equal to zero whenever input voltage or current are equal to zero as well. This zero-crossing property, Chua and Kang write, manifests itself vividly in the form of a Lissajous figure, which always passes through the origin. Thus, they extended the definition of memristor that is now to encompass any system able to demonstrate a Lissajous figure (later called pinched hysteresis loop by Chua) in the  $i$ - $v$  curve, which is presented on Figure 1.




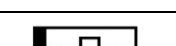
### III. MEMRISTOR MODELS

However, the true interest has been sparked by the notable work of Richard Stanley Williams' group of researchers at Hewlett-Packard laboratories. Despite this fact, the idea of

memristors not being a purely theoretical concept has captivated minds of many researchers around the world, resulting in more than 120 publications about memristors and memristive systems by 2011. [8].

After the concept of memristor was brought back to the public's sight, several implementations of memristors and memristive systems have been proposed. Different implementations of memristor rely on various physical and chemical reactions that give rise to both memristance and nonvolatility, properties essentially constituting the definition of memristor. There have been reported polymeric [9,10], spintronic [11], ferroelectric [12] and layered [13] implementations of memristor, but titanium dioxide memristors remain the most well studied group. During this research four models were closely considered, namely linear ion drift model[1], nonlinear ion drift model[14], Simmons tunnel barrier mode[15], and threshold adaptive memristor model (TEAM)[16]. Unfortunately, due to the paper size considerations only the last one of them will be reported. This model, however, was decided to be further utilized throughout the work.

TABLE I. DIFFERENTIAL EQUATIONS OF BASIC CIRCUIT ELEMENTS

Device	Electronic Symbol	Unit	Differential equation
Resistor		R, ohm	$R = \frac{dv}{di}$
Capacitor		C, farad	$C = \frac{dq}{dv}$
Inductor		L, $\frac{Wb}{A}$ or henry	$L = \frac{d\varphi}{di}$
Memristor		M, $\frac{Wb}{c}$ or ohm	$M = \frac{d\varphi}{dq}$

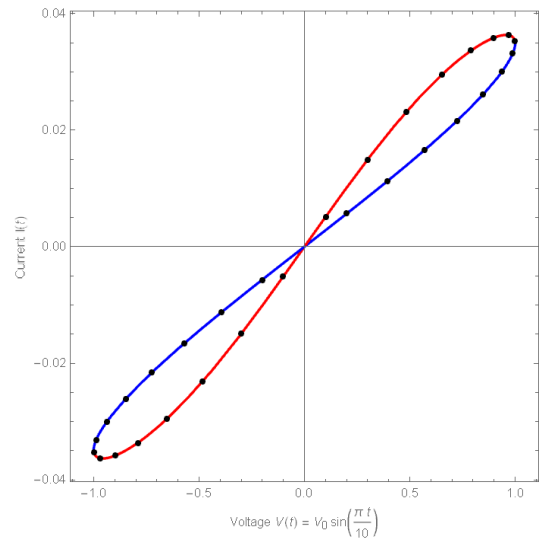


Fig. 1. Pinched hysteresis loop in the  $i$ - $v$  curve

TEAM model, proposed by Kvatinsky et al., incorporates advantages of ion drift models' explicitness and Simmons tunnel barrier accuracy, yet manages to preserve relatively high computational performance and generalizability. TEAM model is based on the same physical behavior as Simmons tunnel barrier model. But it manages to convey it with simpler mathematical functions. The model introduces several assumptions for the sake of analytical simplicity: state variable does not change below a certain threshold and exponential dependence is replaced with a polynomial one. Detailed mathematical foundation of the model may be found in the corresponding paper.

A major advantage of such a relation is the explicitness of current and voltage relationship as opposed to the Simmons tunnel barrier model. Nevertheless, Kvatinsky et al. were able to perform a fitting procedure forcing TEAM model to match the latter with reasonable and sufficient accuracy. In their paper, authors of TEAM model also report the results of comparison between the fitted TEAM and Simmons tunnel barrier model. The feasible preciseness of TEAM model was proved by the average discrepancy between models' state variable difference of only 0.2%. The maximum difference of this value constituted 12.77%, however the run time of the model was nearly halved (47.5%) Kvatinsky et al. had been also able to fit the model with different types of physical memristor models, namely STT-MRAM and Spintronic memristors.

#### IV. MEMRISTOR BRIDGE NEURAL NETWORK

This paper considers the neural network architecture proposed by Adhikari et al. in 2012 [4]. The architecture is based on the memristor-bridge synapse [17] and aims to solve the issue of nonvolatile synaptic weight storage and implement a newly proposed hardware learning method.

##### A. Memristor Bridge Synapse

Memristor bridge synapse architecture was first proposed in [17], it is a Wheatstone-bridge-like circuit that consists of four identical memristors of opposite polarities. When positive or negative strong pulse  $v_{in}(t)$  is applied at the input, the memristance of each memristor is increased or decreased depending upon its polarity.

Kim et al. write, that if input pulse voltage is equal to  $v_{in}$ , voltages at memristors can be calculated according to "voltage-divider formula". Then given memristances  $M_1, M_2, M_3$ , and  $M_4$  stand for the corresponding memristors at time  $t$ , the output voltage is reported to be equal to the voltage difference between terminals A and B:

$$v_{out} = v_A - v_B = \left( \frac{M_2}{M_1 + M_2} - \frac{M_4}{M_3 + M_4} \right) v_{in}.$$

##### B. Memristor Bridge Neuron

In artificial neural networks neurons are required to sum a set of input postsynaptic signal and, according to the activation function, propagate (or not propagate) the signal further on to the next layer of the network. The neuron is then required to sum the input postsynaptic signals. Kim et al. point out, that the signal summing operation is easier to be performed in current

mode: postsynaptic signals should be connected to a single node, so that the following neuron would receive the sum of currents via Kirchhoff's current law. In order to achieve current summation, the memristor bridge synapse has to be modified because it provides voltage output. Kim et al. suggest combining the memristor bridge with differential amplifier. The latter converts post-bridge negative and positive voltage into corresponding currents. Hence, for a set of synapses there exist two nodes: one for positive postsynaptic current and one for negative postsynaptic current. These nodes sum the output currents of each individual synapse in the set. Neuron itself is then comprised of the summation nodes, but also of the active load circuit that implements the activation function as in Figure 2. The sum of all postsynaptic currents is converted back to voltage (presynaptic signal for next layer of neural network) by the active load circuit according to the activation function.

In their paper, Adhikari et al. also provide rigorous mathematical explanation of the suggested architecture behavior.

##### C. Neural Network Training

A composition of an arbitrary number of neurons connected via memristor-bridge synapses therefore constitutes the artificial network. Adhikari et al. intend to use Chip-in-the-Loop technique for training the network of proposed architecture. They, however, suggest modifying this technique slightly in order to take into account peculiar properties of memristor-based circuits. This technique is a viable choice since it provides a way to deal with memristor bridge non-idealities without explicitly modelling these nondidealities. According to this technique, the circuit performs the forward computation of the network, whereas back-propagation and weight update is done on the software side.

The hardware circuit network is reproduced by a software clone, which is used to process the training data. After the computer network has processed all the training data, synaptic weights of each individual synapse circuit are programmed by direct application of strong voltage pulses in order to match with the weights from computer network's weight matrix. Hence, the whole of the hardware network is treated as it consists of a set of simple single-layer networks. Each one of those single layer networks is trained separately, according to the weight matrix.

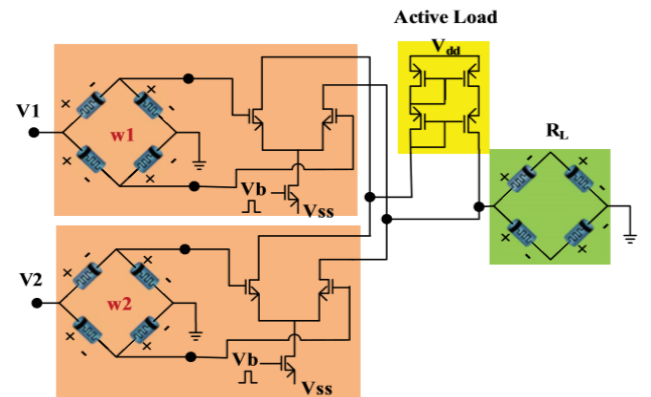


Fig. 2. Memristor Bridge Synaptic Circuit [4]



Because of the nature of memristor bridge synapses, the need in additional circuitry is eliminated.

## V. FRAMEWORK CONCEPT

As one can see, plenty of research has been carried out in the field of memristors and memristor-based neural networks. Multiple approaches to both creating and modelling memristors have been mentioned in previous sections.

It is needed to create a reliable framework for simulating memristor-based neural networks. So far, rather abundant overview of the domain has been presented. Despite the vast variety of works mentioned, the domain at hand lacks general integrity and is not formalized enough to start composing the framework at least in its basic form. Hence, the domain must be formalized to a certain extent. In order to derive this degree of formalization, the requirements for the stated framework are to be determined. This will enable framework to be designed properly and will ensure it complies with the needs and wants of its users. Requirements are decided to include four major points: accuracy, performance, flexibility, and explicitness. Accuracy stands for reliability of framework and if its output data can be trusted. Performance reflects how quick does the simulation proceed. Flexibility corresponds to how easy it is to swap components and models within framework. Finally, explicitness is determined by the overall convenience of the framework and how well does it represent results of the simulation. Insights into these requirements can be better revealed according to the SMART criteria (a project management technique for elaborating objectives), which is done in Table II.

The requirements described above help determine what is to be expected from the framework, what kind of formalization for the domain is required, and set guidelines for further process of

design and development. The domain may be formalized by representing it as a graphical scheme, henceforward called ontological model. The reason for such naming is that this model encompasses relevant entities of the domain under discourse, as well as reflects their major properties and interrelations, which in turn roughly corresponds to the definition of ontology. This model will limit the complexity of the field of memristor-based neural networks and expose the intrinsic connections between the notions at hand.

First, let us derive a set of entities to be found within this model. At the very core of every network there are neurons and synapses. These three notions (neural network, synapse and neuron) constitute the heart of designed model as well.

Multilayer network usually distinguishes between input layer neurons, output layer neurons and hidden layer neurons, which may slightly differ. Input neurons should be able to receive input signals, which may not necessarily coincide with how the signals are conveyed within the network. Similarly, output neurons must provide output signals. Consequently, input and output program modules should be introduced, in order to convert electrical output signals into human-comprehensible format and vice versa for the input signals.

Both neurons and synapses of hardware neural networks are implemented through circuits. Circuit design may vary from one implementation to another, therefore, the general concept of neurons and synapses should be decoupled from its' particular hardware implementation to ensure flexibility. This will enable the framework to safely switch between specific circuit implementations of neurons and synapses, but will also ensure framework's operability. The framework must as well be able to switch between different realizations of memristor, namely, memristor models. Hence, the latter should be considered a

TABLE II. FRAMEWORK REQUIREMENTS ACCORDING TO SMART

Criterion	Accuracy	Performance	Flexibility	Explicitness
Specificity	Results of simulation within framework must coincide with corresponding experimental data.	Simulation processing must be performed in a reasonable time.	Frameworks components must be easy to change and replace, due to the domain's novelty.	Simulation results should be clear and easy to observe.
Measurability	Given the same input data the framework must produce the same output data as in either experimental data or in verified models. Thus, the discrepancy between these results may be used to measure accuracy of the framework.	Time taken to perform the simulation and calculate the results reflects how well does the framework perform in terms of performance.	Framework's flexibility can be measured in regard with how many approaches to memristor modelling and network training and architecture does it implement.	Explicitness is the most subjective of all requirements and should be estimated by direct responses of framework's users.
Achievability	Accuracy is achieved through testing the framework and tuning it match with known data.	Performance is achieved through optimization of frameworks algorithms and architecture.	If designed correctly the architecture (structure) of the framework should provide sufficient flexibility.	Various parameters of framework's components must be accessible for the user. Framework should also provide visualization methods (graphs, visual models, etc.)
Relevance	Accuracy is arguably the most important requirement, without sufficient accuracy, the purpose of the framework is defeated.	Performance is quite relevant since long runtime may hinder the research progress when using framework.	Because the domain is so new, it is extremely important to make the framework able to adapt to possible changes.	Visual representation of simulation results is very important for the end user.
Timeliness	Accuracy may be achieved after tuning the initial version of framework.	Performance should be taken into account during the development, but can be also improved by later optimization.	Flexibility must be ensured from the very beginning of the development.	Visualization may be introduced after the basis of the framework is complete.



separate entity, which is contently used as a component in synapse circuitry. For the time being only the metal dioxide class of memristors is considered to limit already reasonable complexity of the framework.

Finally, the network must should be able to employ different learning techniques. Despite the fact that this work considers only chip-in-the-loop method, the framework should be designed being able to implement various ways of network training. Here it is necessary to take into account not only the learning algorithm, but also how this algorithm is applied to hardware circuit components of the network.

The ontological model is depicted on Figure 3. Solid border circles correspond to the entities of the domain; dashed border circles stand for the properties (attributes) of certain entities; filled arrows represent association relation between entities; empty arrows reflect inheritance (or, possibly, interface implementation); finally, dashed lines reflect attribution connections.

It must be noticed, that the ontological model is likely to be changed in the following works and presented version is not final. Some of the anticipated issues include particular implementations of learning techniques, for instance, chip-in-the-loop does not require auxiliary circuitry, whereas spike timing-dependent plasticity usually does. Another bottleneck to be expected relates to the circuit implementations of neurons and synapses. The latter may consist of multiple circuits that should be represented as separate entities in order to preserve flexibility of the system, yet should conform to the same interface for the sake of integrity.

In this way we shed light onto the structural peculiarities of the future framework. This model is to help composing the classes to be implemented as well as their interrelations. Let us now consider the other side of the developed system, namely, its

functional requirements. In this paper, the latter refer to a certain number of capabilities expected by users from the framework.

Framework under development strives to model memristor-based neural network suggested by Adhikari et al., which is described in the previous section. It is also expected to make possible modeling with better level of preciseness by enabling swappable memristor models. For instance, employing TEAM memristor model may significantly raise the relevance of proposed hardware neural network model through fostering the accuracy of memristor's physical model.

The functional scope of the framework may be represented as a set of intertwined mathematical equations that describe various parts of the network model. Each entity of the framework can be characterized with equations that have adjustable parameters, which are usually derived by the authors of corresponding models from experimental data analysis. These equations are extracted from relevant models and are bound in such way, that one equation's output usually corresponds to input of the other equation. This set of equations is depicted on Figure 4. Each separate square on the scheme reflects an entity of the framework, while arrows denote the input-output connections between equations. One may notice that relations of equations form a cycle, where one iteration of this cycle corresponds to one layer of hardware memristor network. This figure depicts what set of functions is expected to be provided by the future framework.

### CONCLUSION AND PROSPECTS

In this paper, a range of memristor models has been reviewed together with some of the fundamental papers on memristor-related technologies. Based on this review, a concept of framework for modeling memristor-based hardware neural networks has been proposed. This framework represents an

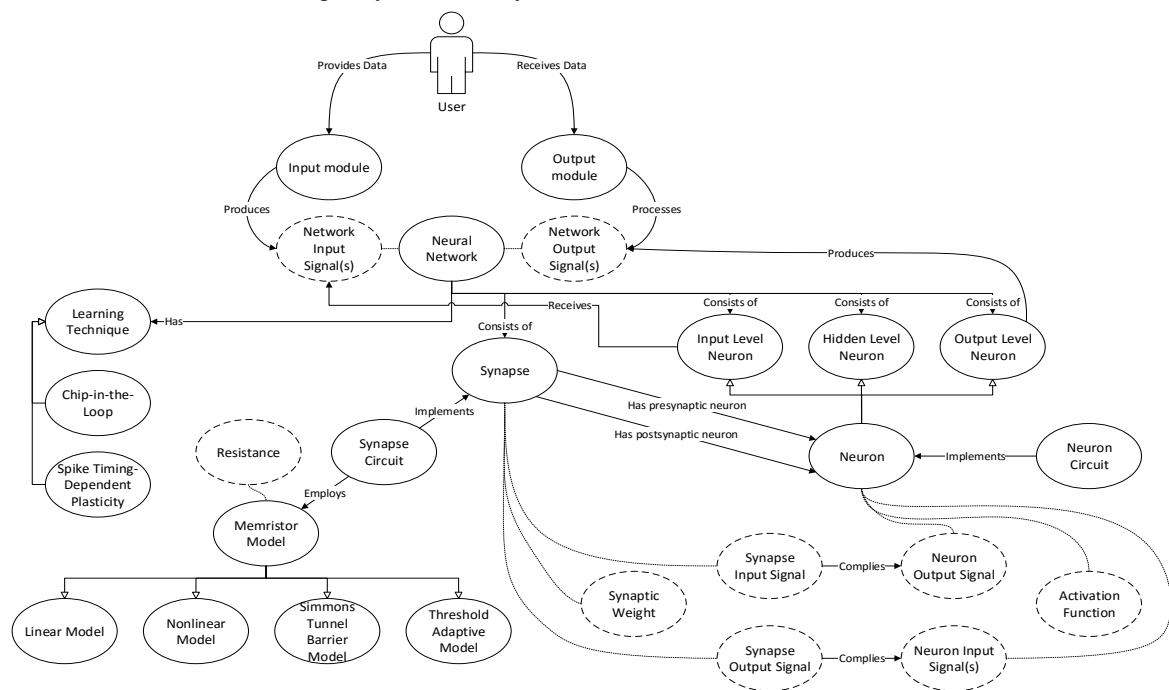


Fig. 3. Domain's Ontological Model

implementation of neural network architecture considered in the paper, but implies ability to swap memristor models in order to increase the overall flexibility and, possibly, relevance of models generated with the help of proposed framework. The ability to switch between model is also expected to help comparing suggested implementations. In the process of framework structure discovery a set of criteria has been formulated to assess the future software product, domain of memristor-based neural networks has been formalized to a certain extent, and, finally, the framework has been given a functional structure strictly defining its' capabilities.

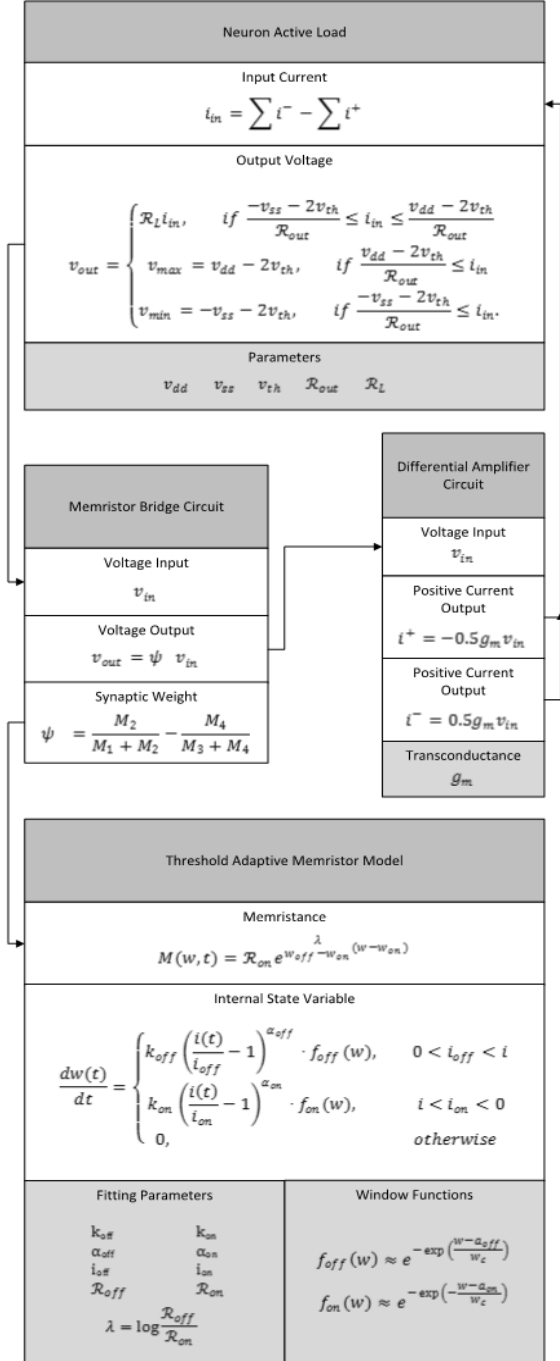


Fig. 4. Functional Structure

Specific platform for framework implementation is yet to be chosen. As of current state of affairs, Unity engine is expected to be the most favorable candidate. Its architecture perfectly fits the nature of soft simulation (which the framework ultimately represents), providing some software patterns that greatly alleviate the development. Considered engine is also able to realize extensive visualization of models as well as equip them with user-friendly interface to further enhance model explicitness and facilitate employment of the future framework for academic purposes. Finally, implementing a circuit simulation framework in Unity also pursues an exploration goal, since such attempts have not been previously well studied.

## REFERENCES

- [1] D. Strukov, G. Snider, D. Stewart and R. Williams, "The missing memristor found", *Nature*, vol. 453, no. 7191, pp. 80-83, 2008.
- [2] J. Mullins, "Memristor minds: The future of artificial intelligence", *NewScientist Magazine*, no. 2715, 2016.
- [3] S. Draghici, "Neural Networks in Analog Hardware - Design and Implementation Issues", *International Journal of Neural Systems*, vol. 10, no. 1, pp. 19-42, 2000.
- [4] S. Adhikari, Changju Yang, Hyongsuk Kim and L. Chua, "Memristor Bridge Synapse-Based Neural Network and Its Learning", *IEEE Trans. Neural Netw. Learning Syst.*, vol. 23, no. 9, pp. 1426-1435, 2012.
- [5] T. Simonite, "A Better Way to Build Brain-Inspired Chips", *Cacm.acm.org*, 2015. [Online]. Available: <http://cacm.acm.org/news/186782-a-better-way-to-build-brain-inspired-chips/fulltext>. [Accessed: 30- Mar- 2016].
- [6] L. Chua, "Memristor-The missing circuit element", *IEEE Trans. Circuit Theory*, vol. 18, no. 5, pp. 507-519, 1971.
- [7] L. Chua and S. Kang, "Memristive devices and systems", *Proceedings of the IEEE*, vol. 64, no. 2, pp. 209-223, 1976.
- [8] A. Thomas, "Memristor-based neural networks", *Journal of Physics D: Applied Physics*, vol. 46, no. 9, p. 093001, 2013.
- [9] V. Erokhin and M. Fontana, "Electrochemically controlled polymeric device: a memristor (and more) found two years ago", *Arxiv.org*, 2008. [Online]. Available: <http://arxiv.org/abs/0807.0333>. [Accessed: 30- Mar- 2016].
- [10] F. Alibart, S. Pleutin, D. Guerin, C. Novembre, S. Lenfant, K. Lmimouni, C. Gamrat and D. Vuillaume, "An Organic Nanoparticle Transistor Behaving as a Biological Spiking Synapse", *Adv. Funct. Mater.*, vol. 20, no. 2, pp. 330-337, 2010.
- [11] X. Wang, Y. Chen, H. Xi, H. Li and D. Dimitrov, "Spintronic Memristor Through Spin-Torque-Induced Magnetization Motion", *IEEE Electron Device Lett.*, vol. 30, no. 3, pp. 294-297, 2009.
- [12] A. Chanthbouala, V. Garcia, R. Cherifi, K. Bouzehouane, S. Fusil, X. Moya, S. Xavier, H. Yamada, C. Deranlot, N. Mathur, M. Bibes, A. Barthelmy and J. Grollier, "A ferroelectric memristor", *Nature Materials*, vol. 11, no. 10, pp. 860-864, 2012.
- [13] A. Bessonov, M. Kirikova, D. Petukhov, M. Allen, T. Ryhnen and M. Bailey, "Layered memristive and memcapacitive switches for printable electronics", *Nature Materials*, vol. 14, no. 2, pp. 199-204, 2014.
- [14] E. Lehtonen and M. Laiho, "CNN using memristors for neighborhood connections", 2010 12th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA 2010), 2010.
- [15] M. Pickett, D. Strukov, J. Borghetti, J. Yang, G. Snider, D. Stewart and R. Williams, "Switching dynamics in titanium dioxide memristive devices", *J. Appl. Phys.*, vol. 106, no. 7, p. 074508, 2009.
- [16] S. Kvatinisky, E. Friedman, A. Kolodny and U. Weiser, "TEAM: Threshold Adaptive Memristor Model", *IEEE Trans. Circuits Syst. I*, vol. 60, no. 1, pp. 211-221, 2013.
- [17] H. Kim, M. Sah, C. Yang, T. Roska and L. Chua, "Memristor Bridge Synapses", *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2061-2070, 2012.

# A Method of Converting an Expert Opinion to Z-number

Glukhoded Ekaterina<sup>#1</sup>, Smetanin Sergey<sup>#2</sup>

<sup>#</sup>*Faculty of Computer Science, National University Research University Higher School of Economics  
Moscow 101000, Russia*

<sup>1</sup>glukhodedkate@gmail.com

<sup>2</sup>sismetanin@gmail.com

## *Abstract.*

The concept of Z-numbers, introduced by Zade in 2011, is discussed topically nowadays due to its aptitude to deal with nonlinearities and uncertainties whose are common in real life. Z-numbers have a significant potential in the describing of the uncertainty of the human knowledge because both the expert assessment and the Z-number consists of restraint and reliability of the measured value. In this paper, a method of converting an expert opinion to Z-number is proposed according to set of specific questions. In addition, the approach to Z-numbers aggregation is introduced. Finally, submitted methods are demonstrated on a real example.

of restraint and reliability of the measured value. In this paper, the method of converting an expert opinion to Z-number is proposed and the new aggregation approach is introduced. At the end, suggested methods are demonstrated.

The paper is organized as follows. In section 2 required preliminaries are presented. In section 3 problem statement is described in details. In section 4 a method of converting expert assessment to Z-numbers is proposed. In addition, the approach to Z-numbers aggregation is introduced. In section 5 proposed methods is demonstrated on the real-life example. In the last section the key results of the article is mentioned and further ways of research is suggested.

## I. INTRODUCTION

Science and engineering tends to deal with different kinds of measures and evaluations, but in fact not all assessment of information can be represented as a clear number. It's common practice for human beings to describe the information in a linguistic terms which are more convenient in everyday life but unsuitable for a standard mathematical representation. In this case information seems to be approximate because usually people assigns a different degree of the certainty depending on circumstances and the context of the data.

In order to resolve problem of the uncertainty degree representation, Zadeh proposed the concept of Z-numbers in 2011[1]. According to this concept, Z-number describes an uncertain variable  $V$  as an ordered pair of fuzzy numbers  $(A, B)$ , where the first number is a fuzzy set of the domain  $X$  of the variable  $V$  and the second one is a fuzzy set that specifies the level of a reliability of the first number as a unit interval.

Fuzzy logic methods are discussed topically last few decades due to its aptitude to deal with nonlinearities and uncertainties whose are common in real life. Despite the widespread application of many methods of fuzzy logic, it seems to be critical to talk about decision appliance without relation to the confidence and the reliability of analysed information especially in the field of fuzzy decision-making. For example, the decision, which was accepted based on low-reliability data, tends to be useless or even harmful on a practice usage. In this case, Z-numbers have a significant potential in describing uncertainty of the human knowledge because both the expert assessment and the Z-number consist

## II. PRELIMINARIES

### **Definition 1:** *A linguistic variable.*

A linguistic variable is a variable whose values are linguistic expression such as sentences, phrases or words in a artificial or natural language. Processing data provided in linguistic variables requires the computing in terms of nonlinear approaches and leads to results, which are also not precise as the original data.

In general, the usage of linguistic variables is motivated by the feature that they provide more generalized information in contrast with numeric variables. For example, Speed is a linguistic variable which can be set to 'very slow', 'slow', 'middle', 'quite high', 'high', 'very high', etc. In natural language this linguistic variable may be represented as follows: 'The speed of the car is slow'. In this case, the characteristic of object under observations given in generalized form i.e. without any specific numeric values, so expert has no need in specific measuring equipment for object estimation.

### **Definition 2:** *Fuzzy sets [2].*

Let  $X$  be a space of points (objects), with a generic element of  $X$  denoted by  $x$ . Thus,  $X = \{x\}$ .

A fuzzy set (class)  $A$  in  $X$  is characterized by a membership (characteristic) function  $\mu(x)$  which associates with each point in  $X$  a real number in the interval  $[0, 1]$ , with the value of  $\mu(x)$  at  $x$  representing the 'grade of membership' of  $x$  in  $A$ . Thus, the nearer the value of  $\mu(x)$  to unity, the higher the grade of membership of  $x$  in  $A$ . When  $A$  is set in the ordinary sense of term, its membership function can take on only two values 0

and 1, with  $\mu(x)$  reduces to the familiar characteristic function of set  $A$ .

In the decision-making tasks each expert gives his own opinion and then it is needed to represent given information in a form that can be processed by a machine. We can use fuzzy numbers for representing the information. Fuzzy numbers can be defined as follows.

**Definition 3:** A *fuzzy number*.

A fuzzy number is a convex and normalized fuzzy set with membership function, which is defined in  $\mathbb{R}$  and piecewise continuous. In other words, a fuzzy number represents an interval of crisp numbers with fuzzy boundary.

Classical example of fuzzy number is triangular fuzzy number. It is represented by a set of two boundary points  $a_1, a_3$  and a peak point  $a_2$ , i.e.  $[a_1, a_2, a_3]$ , as shown in Fig. 1.

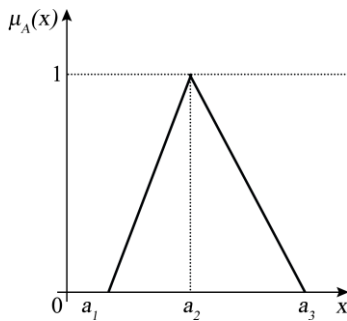


Fig. 1 A triangular fuzzy number

A concept of Z-numbers was proposed in 2010, which was associated with a factor of information reliability for decision-making tasks, a description of the various aspects in the world, an expression of ideas or assessments.

**Definition 4:** A *Z-number* [3].

A Z-number is an ordered pair of fuzzy numbers denoted as  $Z = (A, R)$ . The first component  $A$ , a restriction on the values, is a real-valued uncertain variable  $X$ . The second component  $R$  is a measure of reliability for the first component.

**Definition 5.** A *Z+-number*.

Z+ number is a combination of fuzzy number,  $A$ , and a random number,  $R$ , written as an ordered pair  $Z+ = (A, R)$ .  $A$  plays the same role as in a Z-number,  $R$  is a probability distribution of random number.

### III. PROBLEM STATEMENT

Communications between people is often reduced to the expression of their opinions, reviews or evaluations. Some examples of everyday expert assessments are follows:

- (i) «What is the weather forecast for tomorrow? I really don't know, but I am quite sure that it will be warm» In this example, during the conversation an expert provides an assumption of the prospective weather in linguistic terms and mention a degree of confidence in it. Therefore,  $X =$  Weather forecast for tomorrow, and  $Z = \langle \text{warm, quite sure} \rangle$ .

- (ii) «It takes me about 2 weeks to finish course work.»

Therefore,  $X =$  Time to finish course work, and  $Z = \langle \text{about 2 weeks, usually} \rangle$ .

Generally, the formalization of the statements in a natural language is a complex and unobvious task. For example, the degree of confidence or reliability of the expert estimation can be provided in two ways, namely in explicit or implicit form. The explicit form is represented in example (i) in linguistic term 'quite sure' and the implicit form is contained in the context in example (ii).

Now, it is needed to formulate the problem in a general way. Consider, set of objects ( $\Omega$ ) that needs to be assessed by experts or people who have specific knowledge in the field that relates to  $\Omega$ . Also, there is set of criteria ( $\mathcal{A}$ ), that should be taken into account. In this paper it is not supposed to describe the methods for assessments aggregation. That is why it does not need to involve set of experts in problem statement formulation.

Each expert expresses his own opinion by filling the form with question that are written in a developed form. Questions are formulated using conventional language, such as "how can you assess the level of safety of the given system?" or "Are you sure that the level is high?" or, probably, the most complicated: "How can you assess the distribution of that parameter? Is it Gaussian?" Strict form will be illustrated in the chapter V.

When expert filled his questionnaire with answers it is needed to represent given information in a form that can be processed by machine. There could be several levels of abstraction for representing the information.

TABLE I  
LEVELS OF ABSTRACTION

Level	Numbers	
3	Z-numbers or Z+-numbers	
2	Fuzzy numbers	Random numbers
1	Intervals	
0	Crisp numbers (Integer or Float)	

In this paper the highest level supposed to be considered – Z-numbers. Actually, there will be even Z+-numbers in the chapter IV and V.

The main goal of all paper is to describe the recipe of how human-readable information could be represented as Z-numbers. It should be also mentioned that expert's opinions per each criterion should be somehow accumulated (or aggregated) in a common Z-number that will describe whole relation of a criterion to the considered object.

### IV. THE PROPOSED METHOD OF CONVERTING AN EXPERT OPINION TO Z-NUMBER

First of all, it is needed to describe strict form that expert needs to fill in order to provide knowledge. Form should contain  $N$  sections of  $\mathcal{A}$ . Each section should contains several questions that should describe relation of the criterion  $C \in \mathcal{A}$  to the object  $O \in \Omega$ . Common questions are follows:

1. How does O meet C? Specify the level.
2. Do you have an experience with O? How wide was it?
3. Have you take into account C when using O previously? (If have any experience)
4. Do you follow the latest information about O? When was the latest update?
5. (Only if experts said 'yes' on second question) Which distribution, you think, respects to people perception of C when talking about O?

First and fourth questions are related to the main part of Z-number, other questions are somehow related to the measure of reliability for the first component.

Let us begin with the main part of Z-number. First question is direct. It is supposed that expert assesses the level of affection C on O in the following terms: very high (8, 9, 10), high (7, 8, 9), medium (5, 6, 7), low (3, 4, 5), very low (1, 2, 3), does not meet at all (0, 1, 2). However, it would be incorrect to assign fuzzy numbers to each option before he answered fourth question. This question would show how precise expert could be answering previous one. For example, if he says that he does not follow information for a long time, it should be a clear sign that bounds of each triangle (or trapezoidal) fuzzy-numbers should be widened due to some incompetency in the given field. According to that, there could be different fuzzy numbers for medium: (4, 5, 6) or (2, 5, 8).

The formation of second part should be following. Possible answers for question two are: wide experience, have experience, some experience, little experience, no experience. All these statements could be represented as fuzzy number: (0.8, 0.9, 1.0), (0.65, 0.75, 0.85), (0.4, 0.5, 0.6), (0.1, 0.3, 0.4), (0, 0.1, 0.2). Answers for the question three could be: yes, a lot (0.8, 0.9, 1.0); yes, sometimes (0.5, 0.6, 0.7); yes, a little (0.2, 0.3, 0.4); no (0.0, 0.1, 0.2). Fifth question is auxiliary and will not be converted into fuzzy number. It relates to Z+-numbers. Possible answers: Gaussian, Inverse Gaussian, Binomial, Gamma. It is not prohibited for an expert to skip this question. However, if expert answers the question the specific 'confidentiality' fuzzy number has a value of (0.6, 0.7, 0.8) – 'high', otherwise – (0.0, 0.1, 0.2) – 'low'.

Then several rules should be formulated to construct possibility measure of Z-number:

IF experience='wide' AND take-in-account='a lot' AND confidentiality='high' THEN measure='very high'.

Following table is completed according to the given format.

TABLE II  
RULES FOR PROBABILITY MEASURE

#	CONDITION			RESULT
	Exper.	Take-in	Conf.	Measure
1	Wide	A lot	High	Very high
2	Wide	A lot	Low	High
3	Wide	Sometimes	High	Very high
4	Wide	Sometimes	Low	High
5	Wide	A little	High	High
6	Wide	A little	Low	Medium
7	Wide	No	High	High

8	Wide	No	Low	Medium
9	Have	A lot	High	Very high
10	Have	A lot	Low	High
11	Have	Sometimes	High	High
12	Have	Sometimes	Low	Medium
13	Have	A little	High	High
14	Have	A little	Low	Medium
15	Have	No	High	Medium
16	Have	No	Low	Low
17	Some	A lot	High	High
18	Some	A lot	Low	Medium
19	Some	Sometimes	High	Medium
20	Some	Sometimes	Low	Medium
21	Some	A little	High	Medium
22	Some	A little	Low	Low
23	Some	No	High	Medium
24	Some	No	Low	Low
25	Little	A lot	High	Medium
26	Little	A lot	Low	Low
27	Little	Sometimes	High	Medium
28	Little	Sometimes	Low	Low
29	Little	A little	High	Low
30	Little	A little	Low	Low
31	Little	No	High	Low
32	Little	No	Low	Very low
33	No	-	-	Very low

Then, after the result is given, it should be converted to fuzzy number:

- Very high – (0.8, 0.9, 1.0)
- High – (0.6, 0.7, 0.8)
- Medium – (0.4, 0.5, 0.6)
- Low – (0.2, 0.3, 0.4)
- Very low – (0.0, 0.1, 0.2)

The B-part is given.

The resulting Z-number should be constructed from both parts Z (A, B).

Then, it is time to aggregate different Z-numbers into one Z-number which can describe whole relation of O to the subject according to C based on expert's opinion. There are, at least, three methods of aggregation:

1. Converting Z-number into simple fuzzy number and aggregate them using simple methods.
2. Aggregating A and B-part separately.
3. Converting Z-number into Z+-number, aggregating Z+-numbers and then convert given Z+-number into Z-number.

First approach is the simplest one, but lose some information from an expert. Third approach is one where loss of information is minimized, but it is complicated and it would be difficult to provide all calculations in this paper (requires some non-linear optimization algorithms at some stages). That is why second approach is chosen.

At first stage, it is needed to aggregate A-parts:

$$\tilde{w} = (a_r, b_r, c_r)$$

Where  $a_r = \min(a_i)$ ,  $b_r = \frac{1}{n} \sum_{i=1}^n b_i$ ,  $c_r = \max(c_i)$ , N- total number of Z-numbers. (1)

Aggregation of B-parts are more complicated. First of all, it is needed to multiply one number on another. The resulting formulas are shown at Figure 1.

$$\mu_B(z) = \begin{cases} \frac{-(a_1b_2 + a_2b_1 - 2a_1a_2) + \sqrt{(a_1b_2 - a_2b_1)^2 + 4(b_1 - c_1)(b_2 - a_2)z}}{2(b_1 - a_1)(b_2 - a_2)} & a_1a_2 \leq z \\ \frac{-(c_1b_2 + c_2b_1 - 2c_1c_2) - \sqrt{(c_1b_2 - c_2b_1)^2 + 4(b_1 - c_1)(b_2 - c_2)z}}{2(b_1 - c_1)(b_2 - c_2)} & b_1b_2 \leq z \leq \\ 0 & \text{otherwise} \end{cases}$$

Fig. 2 The result of multiplication of two fuzzy numbers. (2)

Second stage of aggregation assumes that square root calculations should be applied to the given fuzzy number. Somehow, these transformations could be compared with a calculation of geometric mean, when talking about crisp numbers. The resulting formulas are shown at Figure 2.

$$\mu_{\sqrt{x}}(x) = \begin{cases} \frac{x^2 - a}{b - a}, \text{ if } \sqrt{a} \leq x \leq \sqrt{b} \\ \frac{c - x^2}{c - b}, \text{ if } \sqrt{b} \leq x \leq \sqrt{c} \\ 0, \text{ otherwise} \end{cases}$$

Fig. 2 The result of square root transformation of fuzzy number. (3)

The resulting fuzzy number is aggregated B-part.

## V. A NUMERICAL EXAMPLE

Consider following example. It is needed to decide whether it is important to have a new developer in a company or not. To decide more accurately, management of software company introduces a research. Several 'experts' are chosen from different departments. Now, focus on a specialist of marketing. His assessment should be transformed into Z-number for further calculations. It is not necessary for that time to come to a complete conclusion, just focus on how expert's evaluation lead to obtaining Z-number.

### 1. Filling a form

There is a strict form with several questions and two criteria.

Questions for the 1<sup>st</sup> criterion – level of business in the department:

1. How do you think, does the department of software development are filled with work? (very much, much, probably, not so much, no, not at all)
2. How often do you communicate with developers during business tasks? Select from: very often, often, quite often, rarely, not communicate.
3. Did you notice anytime, that deadline was broken due to lack of programmers? Do you pay attention on it? Select from: Notice very often, sometimes notice, I've noticed once, Not at all.

4. Did you follow the news of our developers' team? Do you know most of them? Select from: Yes, communicating each day; Yes, communicating several times a week; Yes, but communicating rarely; No, I'm not.

5. Which distribution, you think, respects to people perception of lack of human resources when talking about new coming developer? (Only if you know)

Questions for the 2<sup>nd</sup> criterion – company's resources:

1. How do you think, does the company have enough resource to hire new employee(s) in a software department? Select from: More than enough; enough; quite enough; probably, not enough; not enough at all.
2. Have you ever been interested in our company's revenue, stock prices etc.? Select from: Yes, I follow all news; yes, sometimes; yes, but rarely; probably, once; no.
3. Have you ever thought about how newcomers can change our budget or resources distribution? How you thought about it when you came? Select from: Yes, thought a lot; yes, sometimes; yes, when I came; No.
4. Do you follow latest news about our state, about our resources distribution on different projects? Select from: Yes, always; yes, sometimes; yes, but rarely; no, I'm not.
5. Which statistics distribution, you think, respects to people perception of our resources when talking about new employees? (Only if you know)

The marketing expert gives answers:

Criterion 1:

1. Not so much
2. Rarely
3. Sometimes notice
4. No, I'm not
5. –

Criterion2:

1. Quite enough
2. Yes, I follow all news
3. Yes, sometimes
4. Yes, sometimes
5. –

### 2. Constructing Z-numbers

According to calculations in chapter IV and table II it is possible to calculate two Z-numbers from his answers.

Z-number from the 1<sup>st</sup> criterion: {(1, 4, 7), (0.2, 0.3, 0.4)}

Z-number from the 2<sup>nd</sup> criterion: {(5, 6, 7), (0.6, 0.7, 0.8)}

Second part of Z-numbers is given by applying corresponding rules from Table II.

### 3. Aggregating Z-numbers

A-part of Z-numbers is aggregated simply by applying formula (1) from IV chapter. The resulting A-part: (1, 5, 7).

B-part of Z-numbers is calculated using formula (2) and (3) from Chapter IV. For simplicity, all calculations would not be provided, only bounds for each of fuzzy number at every step. After multiplication following fuzzy number is obtained:

$$B_m = (0.12, 0.21, 0.32)$$

Then, after applying square root transformation:

$$B_r \approx (0.35, 0.46, 0.57)$$

$B_r$  – resulting  $B$  – part of  $Z$  – number

It should be noticed, that bounds are not lines in this case, they are quadratic functions.

That is why, the resulting  $Z$ -number looks as follows:

$$Z_r = \{(1, 5, 7), (0.35, 0.46, 0.57)\}$$

This  $Z$ -number expresses overall relation to the problem of marketing expert. It could be translated to the normal language such as: “he doubts that software department needs new employee and probably, they do not need, but he is not sure enough”

## VI. PROTOTYPE

The prototype of the  $Z$ -number converting is implemented as a web-service. The example of the converting from Section V is represented in the interactive form. The program is available by the URL <http://fuzzyhse.appspot.com/>.

## VII. CONCLUSION

As a result of this research, a method of converting an expert opinion to  $Z$ -number was proposed. The key problems of  $Z$ -number extraction from natural language statements were discussed and an example illustrating the supposed algorithm were provided. In addition, the new method of  $Z$ -numbers aggregation was proposed and demonstrated on the real example.

The further research will be aimed on improving methods of aggregation in order to obtain a more accurate and reasonable result at the output. The high-quality processing of experts' assessments allows the use of this approach in the real world in order to solve complex problems not only in the business sector, but also in the everyday life. The next step is developing an approach to perform arithmetic operations with observed  $Z$ -numbers. Only after successful finishing of these steps, a complete system for  $Z$ -numbers processing could be built.

## REFERENCES

- [1] L. Zadeh, "A Note on  $Z$ -numbers", Information Sciences, vol. 181, no. 14, pp. 2923-2932, 2011
- [2] L. Zadeh, "Fuzzy sets", Information and Control, vol. 8, no. 3, pp. 338-353, 1965.
- [3] B. Kang, D. Wei, Y. Li and Y. Deng, "A Method of Converting  $Z$ -number to Classical Fuzzy Number", Journal of Information & Computational Science, vol. 9, no. 3, pp. 703–709, 2012.
- [4] Rituparna Chutia, Supahi Mahanta, D. Datta, “Arithmetic of Triangular Fuzzy Variable from Credibility Theory”, vol. 2, August 2011.
- [5] Shang Gao and Zaiyue Zhang, “Multiplication Operation on Fuzzy Numbers”, vol.4, no.4, 2009.
- [6] Palash Dutta , Hrishikesh Boruah , Tazid Ali, “Fuzzy Arithmetic with and without using  $\alpha$ -cut method: A Comparative Study”, vol. 2, March, 2011.
- [7] Marcin DETYNIECKI, “Fundamentals on Aggregation Operators”, 2001.



# Development and research of models of self-organization of data placement in software-defined infrastructures of virtual data center

Irina Bolodurina  
Department of Applied Mathematics  
Orenburg State University  
Orenburg, Russia  
prmat@mail.osu.ru

Denis Parfenov  
Faculty of Distance Learning Technologies  
Orenburg State University  
Orenburg, Russia  
fdot\_it@mail.osu.ru

**Abstract**— This article describes self organizing elements of virtual data centers which allows one to use software-defined infrastructure for deploy applications and services both in a mode infrastructure-as-a Service and Platform-as-Service. In study we developed of model the software-defined infrastructure. We performed an experiment to analyze the productivity of software-defined storage. Our experiment has shown that software-defined storage and scheduling algorithm in software-defined infrastructure placement can gain be obtained growth performance compared with the physical storage and virtual machines. This may be need when storage systems work with high intensities requests.

**Keywords**—cloud computing; computing resources; openflow; software-defined networks; virtual data center; software-defined infrastructure; software-defined storage;

## I. INTRODUCTION

Nowadays, the problem of efficient use of available computing resources in data centers is an actual task. Modern communication technologies have built up an environment for many critical business applications and services based on cloud computing. In this article, we will discuss controls of self organizing elements of virtual data centers which allows one to use software-defined infrastructure (SDI) for deploy applications and services both in a mode infrastructure-as-a Service (IaaS) and Platform-as-Service (PaaS).

The traditional problem modern data centers to ensure quality of service (QoS). In recent years in real data centers to ensure the service level agreement (SLA) used two ways have gained best result in enterprise data centers – virtualization of physical servers and virtualization of network. But these methods are not always sufficiently. In real data centers to ensure the service level agreement all the elements must comply with the following requirements:

1) *Load distribution on the existing computing resources should be with taking into account the type of application, as well as collection of services which requested access.*

*Planning should be done from the point of view of compliance with the SLA.*

2) *Allow resource migration for consolidation virtualized objects of cloud systems and eliminate segmentation of physical resources which occurs in the process of data center operation.*

3) *Allow the users to define and use virtual network functions (VNF) for control and distribute data flows between virtual and physical nodes.*

The requirement 3 may be partially fulfilled by in software-defined networks. Other requirements do not supported in existing commercial and open source cloud platforms [1-2]. Also, none of the known planning algorithms resources in cloud platforms do not possess all these properties in the aggregate [1-4]. For the successful carry out all conditions, need the support of self-organization of resources at all levels of virtual data centers. Implementing such a QoS system is challenging in the current data-center architecture.

To solve this problem, all of the critical elements of the data center should be flexible. This paradigm supports in software-defined infrastructure of data centers. It is a new direction, and therefore is not without drawbacks. The main disadvantage is the bottleneck between the infrastructure levels – storages. With the rapid growth of data centers and the unprecedented increase in storage demands, the traditional storage control techniques are considered unsuitable to deal with this large volume of data in an efficient manner. Existing approaches of virtualization data storages and algorithms data placement either don't consider mapping of all resource types [5,8] or can only be used for a fixed network topology of data center [6,7]. The software defined storage (SDS) comes as a solution for this issue by abstracting the storage control operations from the storage devices and get it inside a centralized controller in the software layer [9].

But the inadequate resource allocation, lack of I/O performance prediction and insufficient isolation are affecting the storage performance in the multi-tenant cloud storage environment. In order to guarantee the quality of service,

---

The research work was funded by Russian Foundation for Basic Research, according to the research projects No. 16-37-60086 mol\_a\_dk and 16-07-01004.

software-defined storage is an effective approach in data centers. However, the lack of intelligence, robustness and self adjustment are blocking the applications and promotions of SDS heavily. This paper focuses on the QoS-Aware I/O resource scheduling problem to build SDI in data centers with high availability, scalability SDS and QoS. For the understanding of all the problems we have built a structural model of software-defined infrastructure.

## II. THEORETICAL PART

The basis of the structural model is software-configurable network, which can be represented as a weighted directed multigraph defined by

$$G = (Nodes, Links), \quad (1)$$

$Nodes = \{Node_i\}_{i=1, \overline{N}}$  – a plurality of network devices (nodes/servers/etc.);  $Links = \{Link_j\}_{j=1, \overline{M}}$  – set of arcs representing a network connection. Each connection is a duplex, so there is an inverse of each arc. The connection point of the arc to the node is a network port of network devices.

Each network device is characterized by the following tuple:

$$Node_i = (L_i, P, C, M, T), \quad (2)$$

$L_i$  – set of arcs emanating from this vertices;  
 $P: L_i \rightarrow Z^+ \cup \{0\}$  – function that characterizes the current delay for each arc;  
 $C: L_i \rightarrow Z^+ \cup \{0\}$  – its current residual bandwidth;  
 $M: L_i \rightarrow Z^+ \cup \{0\}$  – its maximum bandwidth;  
 $T \in \{host, switch\}$  – the type of device. This model provided the difference between end devices and communication equipment.

Structural model of software-configurable infrastructure can be defined as a directed multigraph:

$$SDI = (Seg, Connect(t)), \quad (3)$$

vertices of the graph  $Seg = \{Seg_1, Seg_2, \dots, Seg_s\}$  – a plurality of separate geographically separated segments (autonomous systems), interconnected by global networks;  
arcs of the graph  $Connect(t) = \{(Seg_i, Seg_j)\}$  – directed communication between segments through a global network. To connect segments using gateways and BGP.

The segment  $Seg_k \in Seg$  of the distributed software-defined infrastructure can be described in the form of a weighted undirected multigraph:

$$Seg_k = (Devices_k, Links_k, Flows_k(t)). \quad (4)$$

Vertices of the graph are partition of the set

$$Devices_k = Nodes_k \cup Switches_k \quad (5)$$

$Nodes_k = \{Node_{k1}, Node_{k2}, \dots, Node_{kn_k}\}$  means a set of computing nodes;

$Switches_k = \{Switch_{k1}, Switch_{k2}, \dots, Switch_{km_k}\}$  – means a set switches;

The  $Node_{ki} \in Nodes_k$  including partition of the set

$$Node_{ki} = VM_k \cup Control_k \cup GW_k \cup B_k \cup Stg_k \quad (6)$$

$VM_k = \{VM_{k1}, VM_{k2}, \dots, VM_{kzk}\}$  – virtual machines;

$C_k = \{C_{k1}, C_{k2}, \dots, C_{kzk}\}$  – supervisors OpenFlow;

$GW_k = \{GW_{k1}, GW_{k2}, \dots, GW_{kw_k}\}$  – gateways;

$B_k = \{B_{k1}, B_{k2}, \dots, B_{kb_k}\}$  – balancers switches OpenFlow;

$Stg_k = \{Stg_{k1}, Stg_{k2}, \dots, Stg_{kh_k}\}$  – hardware storages (NAS/SAN).

The  $Switches_k \in Switch$  including switches with and without support OpenFlow.

The ribs multigraph  $Links_k = \{(p_{ki}, p_{kj})\}$  is a two-sided network connections between the ports of the network device, and allows multiple concurrent connections between two devices through different pairs of ports.

Each compute node  $Node_{ki} \in Nodes_k$  has the following parameters and dynamic characteristics:

$$Node_{ki} = (M_{ki}, D_{ki}, Core_{ki}, S_{ki}, m_{ki}(t), d_{ki}(t), s_{ki}^{node}(t)), \quad (7)$$

with  $M_{ki} \in N$  и  $D_{ki} \in N$  – being respectively its RAM size (Mb) and disk capacity (Mb);  $Core_{ki} \in N$  – is the number of its computing cores;  $S_{ki} \in R^+$  – is the relative performance of the cores;  $m_{ki}(t) \in [0, 1]$  и  $d_{ki}(t) \in [0, 1]$  – are RAM and disk relative loads for a computing node at time;  $s_{ki}^{node}(t) \in \{"online", "offline"\}$  – is the node state at time  $t$ .

At each node there is a queue of tasks  $Q_{kij}^{node}(t) = \{Q_{kije}^{node}(t)\}$ . They are used to provide QoS according to the minimum guaranteed bandwidth and guaranteed maximum delay for a given networking.

$$Q_{kije}^{node}(t) = (MinB_{kije}^{node}(t), MaxD_{kije}^{node}(t)), \quad (8)$$

$$MinB_{kije}^{node}(t) \in N \cup \{0\} \quad \text{and} \quad MaxD_{kije}^{node}(t) \in N \cup \{0\}$$

represent respectively the minimum bandwidth (in kb / s) and maximum delay for the corresponding port queue (in ms), which were established to ensure QoS mechanism.

The other elements of the infrastructure also have detailed specifications, but in this paper we will not consider them. Let more detail on the data storages.

Hardware networked storage devices contains images of instances of virtual machines, database applications, as well as infrastructural components of cloud computing.

Earlier we considered solutions for cloud data storage, providing data migration, as well as algorithms for data placement on devices [10-11]. Given the new paradigm of software-defined storage facilities, proposed concepts need to be improved in terms of defining the types of data to be placed on their structure. In a study we carried out the classification of the data to be placed on the devices in a network environment. On the basis of the matrix of correspondences we have constructed an algorithm that allows to determine the structuring of the data to determine the final placement process (storage / sql / nosql), the type of physical device (HDD or SSD), as a direct choice of the most appropriate device. Thus, placed in the data can be represented as a structure where *TypeS* storage - method of placement, *TypeD* - view of the physical device, *RDisk* - a physical storage device.

Each segment  $Seg_k$  has the following storage  $Stg_{ki} \in Stg_k$  parameters and the dynamic characteristics

$$Stg_{ki} = (MaxV_{ki}, P_{ki}^{stg}, Vol_{ki}(t), \bar{R}_{ki}(t), \bar{W}_{ki}(t), s_{ki}^{stg}(t)), \quad (9)$$

$MaxV_{ki} \in N$  - the maximum storage capacity (Mb);  $P_{ki}^{storage} = \{p_{kij}^{storage}\}_j$  - its set of network ports;  $Vol_{ki}(t) \in N \cup \{0\}$  - available storage capacity in Mb at a time  $t$ ;  $\bar{R}_{ki}(t)$  и  $\bar{W}_{ki}(t)$  - respectively, the average established by the time of reading and writing speed;  $s_{ki}^{storage}(t) \in \{"online", "offline"\}$  - storage state at time  $t$ .

The study found that the placement of data on the physical devices has a number of limitations that affect both the performance of the operation of reading / writing data, and the process of optimizing the location of data on the devices. To neutralize this limit is proposed to apply the software-defined storage. This type of storage can be represented by the following structure:

$$SoftStg = (Vm, Lan, Stype, Dtype, RDisk(t), Vdisk) \quad (10)$$

*Vm* - the virtual machine or network container, *Lan* - speed network access interface, *Stype* - supported method of data placement, *Dtype* view of the physical device on which the virtual machine will be placed; *RDisk*( $t$ ) the specific physical device containing the virtual machine at the time  $t$ ; *Vdisk* - the total amount of data storage. The advantage of this is the possibility of placing the data migration between the physical storage devices. In this regard and availability of data from the cloud-based platform are continuous, as the connection is made with a virtual repository, rather than a physical device.

The concept of software-defined storage is built on the same principles of self-organization on the basis of abstractions. The present study used a model of computing resources, cloud system developed previously [10]. In addition to the existing facilities in the cloud infrastructure model introduced the concept of an agent and a control unit. An agent is a compute node cloud system that can act both as a computing node *Snode*, storage *Sstg*, network storage *Snas*. Thus at any time, the agent can become the control node. This is due to the clustering of computing nodes.

The basis of the self-organization software-defined storage model laid adaptive dynamic reconfiguration change adaptation resources. This helps to optimize the organizational structure of the cloud platform, namely the search algorithms of optimal control units, as well as the allocation of management groups. Our proposed control model consists of two parts, components and resources. In the formation of software-defined storage per virtual compute node runs software that is responsible for the exchange of technical data on the devices. This exchange is carried out within a group of units engaged in storage with a single storage method. Moreover, among the group of nodes selects the least loaded node, acting as a control unit. This approach reduces the risk of degradation of the control unit during operation. However, if there is a loss of communication with the control unit, the remaining group of virtual machines are always present data to each other, allowing you to make an automatic selection of a new control unit and delegate authority, which also reduces the risk of failure of the control system. In addition to the problem of organizing the exchange and storage management of the group, the control unit interacts with the control units of other groups to maintain current information on the status of all system as a whole. Thus, the entire system of software-defined storage facilities built on the principle of hierarchical network comprising three basic levels: the level of online access, the level of control in the group and the level of exchange of data at the level of the whole system. On the basis of the concept described software-defined storage facilities we implemented the algorithm data placement in software-defined storage.

The basic difference of the structural model of software-defined infrastructure is that in addition to the standard elements of heterogeneous cloud platform, which are integral parts of the notions of cloud applications and services. Each of the applications is a weighted acyclic directed dependency graph by in which the vertices are the cloud server database, storage and other storage and caching resources, arc -

depending on the data between the nodes. Each node-task characterized imposes resource requirements (to the number of cores, architectural teams nuclei size memory, and disk, the presence of special libraries or hardware on the physical or virtual nodes are used to run processes), the number of running processes, the measurement of time, communication patterns transferring data between processes. The originality of the model lies in the fact that each arc is characterized by the type of access (access to the file in the storage system to a local file, a distributed database, a data service and etc.). Estimate the volume of transmitted data, the QoS requirements.

Cloud service, as well as cloud application, described as orient rowan graph data dependencies, the difference lies in the fact that from the point of view of the user cloud service is a closed system. Also, all his applications shared between virtual machines or physical servers pre-set, new instances of them scaled dynamically depending on the number of incoming requests to perform the functions of a particular service from the cloud application, end user or other cloud services. The figure shows the structure of the physical computing node, which is an element of a heterogeneous cloud platform.

### III. PRACTICAL PART

The competitive advantage of the developed control algorithm software-defined storage compared to existing analogues is heuristic analysis of new data types in the process of downloading files in the cloud platform. Thus through storage virtualization is performed transparently to the client mirroring data on multiple storage devices, that provides an increase in speed of data allocation in terms of maintaining the integrity and redundancy. Formation of the self-organizing software-defined storage on the basis of virtual machines and containers can not only reduce the risks associated with the loss of inaccessibility of data, but also provides intelligent analysis of demand data, which are formed on the basis of the card placement of virtual machines and containers. The algorithm is based data placement in software-defined storage facilities on a model that allows describing the structure and connections of virtual devices, machines and containers of data. The model is based on multi-agent approach in the organization of storage. The agents collect system status. This information is analyzed with the use of machine learning algorithms (Data Mining). The output of the analysis is obtained map of location devices within the cloud platform tied to physical devices, as well as a map of the demand generated the data. By analyzing the two cards, and the heuristic algorithm predicting cloud management system decides on reconfiguration or moving virtual storage devices, as well as the rotation and redistribution of data between different nodes of the system. Map of the location of dynamic objects are generated by using time interval.

In addition to the designated problem, an algorithm developed by placing the data in the software-defined storage is used for increasing the productivity of the system components of the cloud. Due to the efficient reallocation of data streams between running instances of virtual machines and containers, provided not only have the quality of service,

but also a compact arrangement of the devices [10]. To resolve this issue as one of the elements in the basis of the developed algorithm, the study data placement in software-defined storage facilities used aggressive version of the algorithm Backfill used to optimize the performance of tasks in the Grid [10]. The task of increasing the efficiency of use of available storage devices based on the data received from the agents and management nodes cloud system through the dynamic resource management in conditions of limited consumption of the computing power is relevant for the cloud. This is primarily due to the economic performance of cloud platforms. A generalized block diagram of the algorithm shown in Fig. 1.

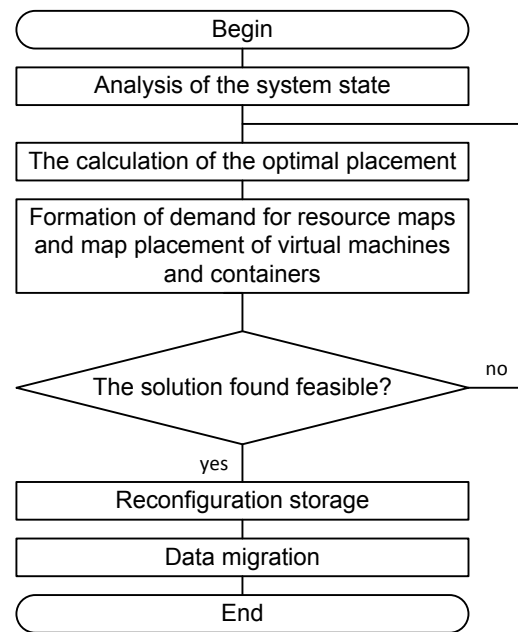


Fig. 1. Base algorithm of data storage organization in SDS

In addition to using elements Backfill algorithm to improve the efficiency of cloud optimization system produced a series of improvements relating to direct data access mechanism. When dealing with services located in the cloud system does not exclude the situation in which the user's request for service may be employed multiple data stores with different access characteristics. When using such data cloud system is necessary to prepare the access time to optimize the reading. To do this, the algorithm developed data placement in software-defined storage in the course of building a series of internal rules, thereby adjusting to the flow of user requests every instance storage. As a result, query execution plans with the same intensity at different times may be distributed differently. Rebuilding of the rules takes place in accordance with the demand for resources to efficiently manage the distribution and dynamic load balancing.

### IV. EXPERIMENTAL PART

We will study workload characteristics, requirement analysis, the theory of QoS in SDS and I/O scheduling strategies. We obtain such goals by execution mechanisms and dynamic robust I/O scheduling algorithms for multi-type

resources allocation. In the current progress, scheduling for software-defined storage has been proposed for the SSD/HDD hybrid storage. The preliminary evaluation in some benchmarks shows that SDS can gain better performance compared with other strategies.

To evaluate the effectiveness of the algorithm data placement in software-defined storage facilities built in the context of models of software-defined infrastructure, we have conducted research work in the cloud system built on the basis Openstack with different parameters. Thus as reference data for comparison in the experiment used standard algorithms used in cloud systems, as well as traditional storage systems. For the experimental study a prototype cloud environment, which includes the main components as well as software modules developed algorithms for modifying the processing of user requests data in a software-defined storages. Thus as reference data for comparison in the experiment used standard algorithms used in cloud systems, as well as traditional storage systems. For the experimental study a prototype cloud environment, which includes the main components as well as software modules developed algorithms for modifying the processing of user requests data in a software-defined storage.

The OpenStack cloud system implemented module that applies an algorithm developed by placing the data in the software-defined storage facilities for the management of computing resources and cloud system efficient allocation of virtual machines to physical hosts, as well as related data. In an experiment designed to analyze the data stream of requests, similar to real traffic cloud infrastructure based on the data log records access to certain types of resources, classified by data type and structure of the query. Retrospective reproducible requests amounted to 3 years, while for load experiment used the averaged data. The data are distributed to a pool of virtual machines on the following criteria: the type of customers to make an appeal to the data, the type of service demanded by the connection. The number of simultaneous requests to the system was 100,000, which corresponds to the maximum number of potential users of the system.

All queries created reproduced consistently at three pilot sites. This restriction is due to the need to compare the results with the physical storage systems are not capable of reconfiguration. The main difference is the use of experimental sites SSDs.

In addition to the platforms to analyze the effectiveness formed 3 groups of experiments aimed at intensive operation for reading (Experiment 1), write (experiment 2) and concurrent read and write data (Experiment 3).

TABLE I. THE RESULTS OF EXPERIMENTAL RESEARCH

Experimental playground	Only HDD		
Storage System	Physical storage	Virtual storage	software defined storage
Experiment	(reading/writing/r+w)		
Total requests processed number of requests	65040/ 54054/ 45064	85046/ 86450/ 76054	95064/ 97056/ 86578

Experimental playground	Only HDD		
Storage System	Physical storage	Virtual storage	software defined storage
Experiment	(reading/writing/r+w)		
Average execution time	6,5/ 9,6/ 12,5	4,3/ 5,8/ 7,6	2,7/ 3,6/ 5,5
The maximum amount of data from the reference value (%)	300/ 300/ 300	178/ 155/ 156	156/ 133/ 134
Experiment	Only SSD		
Total requests processed number of requests	78540/ 68054/ 55305	94504/ 86054/ 73080	99153/ 99055/ 89452
Average execution time	4,5/ 5,6/ 9,5	2,3/ 3,2/ 3,6	1,7/ 2,6/ 2,5
The maximum amount of data from the reference value (%)	300/ 300/ 300	168/ 145/ 134	146/ 118/ 128
Experiment	Hybrid (HDD + SSD)		
Total requests processed number of requests	69480/ 78054/ 67302	92308/ 92450/ 96054	98504/ 97056/ 98778
Average execution time	5,5/ 8,6/ 10,5	3,3/ 4,8/ 7,6	2,2/ 3,6/ 2,6
The maximum amount of data from the reference value (%)	300/ 300/ 300	164/ 164/ 145	124/ 128/ 119
The overall efficiency	50%	64%	87%

The experiment was one hour corresponding to the longest period of time of peak load, recorded in real traffic. After analyzing the data of experimental studies proved that the software-defined storage more efficient, regardless of the type of physical devices. The findings support the use of the algorithm and software-defined storage to provide efficient services to the cloud. The results of the experiments can be concluded to reduce by 20-25% the number of failures in service when placing the data in the software and message-driven data warehouses. In addition, in a pilot study assessed the amount of storage used for experimental platforms 2 and 3, as for the physical storage resources do not support scaling in real time. Due to the optimal allocation of resources for each compute node is guaranteed to work together to ensure all running instances of the application that meets the requirements of potential users. At the same time thanks to the work of the algorithm data placement in program-controlled vaults of opportunity of the release of 20 to 30% of the resources allocated computing nodes. Thus, the proposed algorithm can be used for any computing system architectures, including inhomogeneous physical node configuration and VMs.

## V. CONCLUSION

Thus, assessing the overall result of work the algorithm of data placement in software-defined storage can gain be obtained performance from 20 to 25% compared with the physical storages and virtual machines. This may be need when storage systems work with high intensities requests. In addition, reducing the number of allocated virtual resources allows for more efficient scale cloud systems, and to provide a safety margin with a sharp increase in the intensity of use of the selected applications.

## References

- [1] Bein D., Bein W., Venigella S. Cloud Storage and Online Bin Packing // Proc. of the 5th Intern. Symp. on Intelligent Distributed Computing, 2011, Delft: IDC, P. 63-68.
- [2] Nagendram S., Lakshmi J.V., Rao D.V., et al Efficient Resource Scheduling in Data Centers using MRIS // Indian J. of Computer Science and Engineering, 2011, V. 2. Issue 5, P. 764-769.
- [3] Arzuaga E., Kaeli D.R. Quantifying load imbalance on virtualized enterprise servers // Proc. of the first joint WOSP/SIPEW international conference on Performance engineering, 2010, San Jose, CA: ACM, P.235-242.
- [4] Mishra M., Sahoo A. On theory of VM placement: Anomalies in existing methodologies and their mitigation using a novel vector based approach // Cloud Computing (CLOUD), IEEE International Conference, 2011, Washington: IEEE Press, P.275-282.
- [5] Cheng X., Sen S., Zhongbao Z., Hanchi W., Fangchun Y. Virtual network embedding through topology-aware node ranking // ACM SIGCOMM Computer Communication Review. 2011. V.41. №2. P.38-47.
- [6] Korupolu M., Singh A., Bamba B. Coupled placement in modern Data Centers // IEEE Intern. Symp. on Parallel & Distributed Processing. N. Y.: IPDPS, 2009. P.1-12.
- [7] Singh A., Korupolu M., Mohapatra D. Server-storage virtualization: integration and load balancing in Data Centers // Proc. of the 2008 ACM/IEEE Conf. on Supercomputing. Austin: IEEE Press, 2008. P.1-12.
- [8] Plakunov A., Kostenko V. Data center resource mapping algorithm based on the ant colony optimization // Proc. of Science and Technology Conference (Modern Networking Technologies) (MoNeTeC), Moscow: IEEE Press, 2014. P.1- 6.
- [9] Darabseh, A., Al-Ayyoub, M., Jararweh, Y., Benkhelifa, E., Vouk, M., Rindos, A. SDStorage: A Software Defined Storage Experimental Framework // Proc. of Cloud Engineering (IC2E), Tempe: IEEE Press, 2015. P.341- 346.
- [10] Parfenov, D. Approaches to the effective use of limited computing resources in multimedia applications in the educational institutions / Parfenov D., Bolodurina I. // 2015 5th International Workshop on Computer Science and Engineering: Information Processing and Control Engineering, WCSE 2015-IPCE, 2015.
- [11] Parfenov, D. Approach to the effective controlling cloud computing resources in data centers for providing multimedia services / Parfenov D., Bolodurina I., Shukhman A. // 2015 International Siberian Conference on Control and Communications, SIBCON 2015 - Proceedings, 2015.

# *Automated Text Document Compliance Assessment System*

*Maria A. Zhigalova*

Department of Information Technologies in Business  
National Research University Higher School of Economics  
Perm, Russia  
mariezghalova@gmail.com

*Alexander O. Sukhov*

Department of Information Technologies in Business  
National Research University Higher School of Economics  
Perm, Russia  
ASuhov@hse.ru

**Abstract**—The study is dedicated to the problem of automating an electronic text document compliance assessment in accordance with the formal requirements on formatting set in standards. The need for the software system development of such kind appeared due to laboriousness and inefficiency of manual text check. The system functionality is based on the application of the Open XML SDK solution with the use of FormattingAssembler module included in PowerTools for Open XML. The system provides a comprehensive text document check in accordance with the formatting parameters defined by the user. In practice, the software product can be used to verify compliance with the formal requirements of research papers and dissertations, scientific publications, technical documentation, etc.

**Keywords**—*formatting rules, text document, compliance assessment, DSL*

## I. INTRODUCTION

Text processing, which refers to automation of creation and manipulation of electronic texts, has always been one of the primary disciplines in computer science. It involves determining the quality of publications, identification of potential duplication, plagiarism, partial borrowings, classification and clustering of documents, formation of databases and extensive collections of texts. Despite the fact that document checks in accordance with formatting rules does not imply detailed text processing, it should be noted that this procedure in one way or another is related to general text analysis and has its specific features.

It is known that document checks in accordance with formatting rules is primarily manual. It is considered to be extremely laborious and time-consuming, and researchers, as well as individuals responsible for document check in universities or organizations, are likely to appreciate the simplification of this process. Since the structure of research papers, dissertations, scientific publications, technical documents, etc. is a standard-based compulsory requirement, there is an ongoing need in the instrument allowing users to check the formatting of their work and automatically fix it if necessary. Therefore, the goal is to provide users with such functionality by creating a relevant software product minimizing the time and effort required.

Thus, the focus of the study is on the development of the automated text document compliance assessment system. It is assumed that the software product functionality is extended to the check of such formatting characteristics as page layout settings, styles parameters, headers and footers properties etc. In other words, the document design (not its content) is to be checked. The application can be used by a wide range of users, including students, teachers, technical writers, etc. It is expected, that the automation of text document compliance assessment will significantly increase the efficiency of business processes connected with document check.

## II. RELATED WORKS

To date, there are two basic methods of control of the text on the absence of formatting errors and verification of a document in accordance with certain standards including ready-made design templates and various software solutions.

Violation of document styling often occurs when text is copied into a document from sources with diverse formatting patterns. Although this issue can be partially solved by application of built-in styles, the probability of error still exists. In this case, the use of formatting templates is a reasonable option.

One of the means of creating such templates is a markup language DocBook, which is an application of XML/SGML (XML – eXtensible Markup Language, SGML – Standard Generalized Markup Language). It provides a user with a unified set of tags for setting formatting of a text document [1]. This approach makes it possible to isolate document content from its style representation. The apparent advantage of DocBook is that a predefined set of tags eliminates formatting errors and allows a large number of users to work with the same text simultaneously.

Formatting templates are also utilized by the LaTeX publishing system which provides the capability for automating a process of inputting and formatting text of a document. The content of a LaTeX document, similarly to DocBook, is represented by structural and semantic markup. Text document formatting is described in a separate file with style information [2] which defines formatting rules, specific to each document type. Despite a vast variety of functional characteristics, it should be mentioned that LaTeX has a



number of disadvantages: firstly, in order to manipulate LaTeX documents, it is required to have a special development environment installed on a computer, and secondly, the process of creating a LaTeX document may be challenging for users who are not sufficiently skilled to work with the LaTeX system.

The automation of compliance assessment is implemented in a number of software products, one of which is an intelligent web-based system for spell checking "Orogrammka". The software checks the norms of grammar, punctuation and document formatting [3]. Compliance assessment is provided for research papers and dissertations in accordance with requirements that are set in a number of standards supported by the service. The software has an intuitive and simple interface, however, it should be noted that text check is limited to a strictly predefined set of formatting rules (margin sizes, page layout settings, reference list format, etc.) without the possibility of expanding the functionality by a user.

Another tool for automated formatting rules check was developed in Volgograd State Technical University [4]. This software solution is a Microsoft Word 2007 add-in which allows users to check their documents and fix detected errors. In spite of convenience and ease of use, the service has a significant drawback: users whose personal computers are not running Microsoft Office Word are deprived of the opportunity to perform the compliance assessment of text documents.

Overall, the analysis of the studies mentioned above highlights the need for the software system that provides an extensive functionality for formatting rules check, yet, has a user-friendly interface appealing for a large group of users. This work is to propose such a system.

### III. TEXT DOCUMENT FORMATTING

#### A. Overview and Comparative Analysis of Popular Text Document File Formats

It is known that electronic text documents represent a major part of stored and processed data. This explains availability of a significant number of file formats used for specification of textual information. However, due to the fact that formatting check of various types of text documents requires the use of special software tools, there is a need for selecting the most appropriate file format which is to be used as the basis for the development of a software product.

Thus, the most common editable text file formats were identified:

- OpenDocument Text (\*.odt) – a file format for text documents with an open specification standardised by ISO/IEC 26300; based on XML.
- Rich Text Format (\*.rtf) – a closed cross-platform file format for storing text documents developed by Microsoft. A document with \*.rtf extension consists of commands which can be divided into control words and control characters.
- Microsoft Word (\*.doc) – a proprietary binary text file format used in Microsoft Word 97-2003. Document

files represent complex objects organized according to the rules of structured storage [5]. The basic unit of data measurement is a symbol; all information about characters is in document stream.

- Microsoft Word (\*.docx, \*.docm) – an open file format for storing electronic text documents used in Microsoft Word since version 2007. DOCM extension indicates support of built-in macros and scripts. Microsoft Word with DOCX (DOCM) extension is part of the Office Open XML format. Office Open XML was initially standardized by Ecma-376 and then redefined in ISO/IEC 29500 standard [6]. OpenXML is a structured archived file that contains markup of a document in an XML format, graphical information and other data included in this text document.

Table I contains results of the comparison of electronic text documents formats by a number of parameters that will identify the option most preferred for research purposes.

TABLE I. TEXT DOCUMENTS FORMATS COMPARISON

	OpenDocument Text (*.odt)	Microsoft Word		Rich Text Format (*.rtf)
		*.doc	*.docx (*.docm)	
Date of creation	2005	1997	2007	1982
Open or proprietary	Open	Proprietary	Open	Proprietary
Document file self-sufficiency	Partial	Full	Full	Partial
Ability to convert to other formats	Yes	Partial	Yes (partial for *.docm)	Yes
Free software	Yes	Partial	Partial	Yes
File size compactness	High	Low	High	Low

Unlike Rich Text Format and Microsoft Word (\*.doc), OpenDocument Text and Microsoft Word (\*.docx, \*.docm) file formats have open specifications which allows third-party developers to freely create software for processing text documents with ODT and DOCX extensions. It is also worth mentioning that ZIP archive compression used by these formats significantly reduces file sizes making them more compact.

Microsoft Word documents of all versions are self-sufficient, i.e. they store all necessary data for correct content representation, whereas OpenDocument Text documents may not be displayed correctly in different programs or operating systems and RTF is fully supported only in a limited number of software products. The capability to convert from one format to the other is represented in every case. Comprehensive free software is only available for OpenDocument Text (some features of Rich Text Format are not implemented in freely distributed products). However, it is worth noting that, despite strong connectivity of Microsoft Word to the original Microsoft software and the absence of free alternatives, the usage of the format prevails. Such a conclusion can be drawn

on the basis of statistics from Microsoft [7], according to which about 1.2 billion people around the world use Microsoft Office applications as their primary tool when working with spreadsheets, texts, presentations, etc.

The advantages of a Microsoft Word file format based on an Open XML format [8] include:

1. Interoperability. The capacity of the format to interact and function with a large set of both custom and commercial applications provides a high degree of compatibility of documents for different tasks.
2. Backward compatibility. The ability of transformation of MS-DOC files into Open XML format with high accuracy allows end users to convert these documents to the Open XML format, and then programmatically access the converted documents.
3. Programmability. Minimum requirements for working with Open XML include a tool that can open and save ZIP files and an XML parser/processor. ZIP and XML libraries allow creating documents in Open XML format on a software level.
4. Integration of business data. Office applications support custom XML schemas that can extend the capabilities of the existing Office document types. Thus, users can export data from existing systems to the documents in the Office file formats.
5. Compact file format. Open XML format uses the technology of ZIP compression for storing documents which provide the possibility of reducing storage space. Opening the file causes the automatic unpacking of the archive, and saving the file results in its compressing.

Thus, a comparative analysis of the formats of text documents showed that Microsoft Word (since 2007 version) seems to be the most appropriate option in terms of the use of open standards based on ZIP and XML, the capability of processing in third-party applications, the ability to convert to other formats and popularity among users

### B. WordprocessingML Description

An ISO/IEC 29500 standard specifies a markup language for text document description which is called WordprocessingML. In a WordprocessingML file elements are grouped in accordance with functionality and stored in separate parts of a ZIP archive. For example, information about all footnotes in a document is gathered in one element, however, in case of footers, the situation is slightly different: each section of the document can store up to three different configurations of headers and footers with different numbering options, special first page settings, etc. Thus, the structure of WordprocessingML includes a set of the following elements: a main document, comments, document settings, footnotes, header/footer, styles, fonts table, document glossary, etc. Fig. 1 illustrates parts of a document TestFile.docx opened with a tool Open XML Package Editor PowerTool for Visual Studio that allows to view the file hierarchy of the document archive and the relationships between them and also to modify their markup.

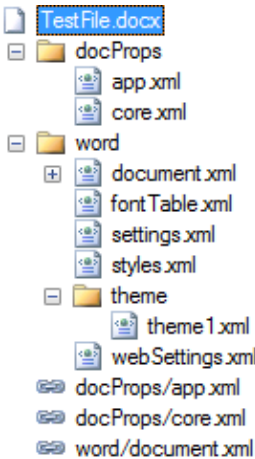


Fig. 1. Microsoft Word document file structure

In the main document part paragraphs (w:p) and tables (w:tbl) can be child elements for document body (w:body), table cell (w:tc) or text box (w:txbxContent). Paragraphs, in their turn, are a run-level content container for text runs (w:r), or images – a VML document (w:pict) or a DrawingML object (w:drawing). Finally, sub-run-level content incorporates multiple text elements (w:t).

Formatting of a text document with the use of Microsoft Word refers to implementation of various styles with parameters included in styles.xml file of a document archive [9]. This file contains data on styles of paragraphs, characters and tables, latent styles and standard settings of styles for an entire document (document defaults). Styles of paragraphs, characters and tables comprise information about current formatting of a document, whereas hidden styles are not used directly and serve primarily as a cache repository for style settings, for example, the ones copied from a template. Standard styles store default values for the entire document formatting. However, it should be noted that styles.xml file does not involve data on formatting of numbered and bulleted lists that is included in a special numbering.xml file.

The fact that content of a document can be formatted on multiple levels leads to a problem of determining a comprehensive set of formatting parameters used for a particular paragraph or a run of the text. These levels of formatting are schematically represented in Fig.2.

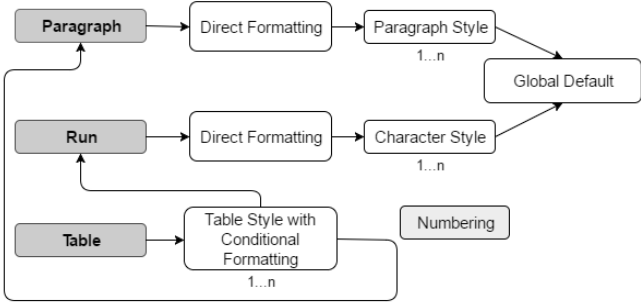


Fig. 2. Levels of Microsoft Word text document formatting

Thus, if it is needed to retrieve information about a paragraph (e.g. line spacing or indentation), the first aspect that has to be checked is direct formatting which is specified in a file called document.xml. Yet, paragraph parameters might not be indicated in this file, and, in this case, it is necessary to inspect the style which is referred to in paragraph properties. If this style does not contain data on the paragraph formatting, then the styles from which it inherits are to be checked. If this action did not bring any results, then the only option left is to process the contents of the node Global default, i.e. default settings of all styles in a document.

Similar approach is credible for checking text runs formatting (defining such font settings as size, name, etc.); the only difference is that character styles are put into consideration and they can also form an inheritance hierarchy.

Data on tables formatting are defined in styles with conditional formatting that specify the properties of rows and columns. Table styles are also inheritable. Text inside table cells is checked according to algorithms of determining formatting of paragraphs and runs. In case of numbering, each list item may include formatting from a paragraph, a numbering format in numbering.xml or a style that is indicated by this format.

Overall, the major difficulty of text document formatting check lies in determining precise formatting parameters for paragraphs, tables, numbering and runs of text for the purpose of conducting as extensive an analysis of conformity of a document to specified rules as possible.

#### IV. SYSTEM DEVELOPMENT

The compliance assessment procedure can be described as follows: the system sequentially retrieves formatting data from document markup and compares it to formatting parameters specified by the user. In order to work with WordprocessingML markup, it was decided to use Open XML SDK 2.5 for Microsoft Office. Retrieval of information on document formatting was performed by using the FormattingAssembler module which is a part of PowerTools for OpenXML. This module accesses style information on every level of formatting and assembles it, so that the markup of an original document is modified in a way that there is only direct formatting left. However, this direct formatting contains all formatting parameters (even from hidden styles) that were applied to a document.

OpenXML SDK built on the System.IO.Packaging API allows users to manipulate documents that adhere to the Office Open XML File Formats Specification, e.g. documents created with Microsoft Office applications. This package provides a set of strongly-typed classes to obtain data about the formatting of a document and makes it possible to modify an original document (for example, to add comments).

Despite the fact that .NET offers standard assemblies for working with Microsoft Office, the preference was given to OpenXML SDK. COM Interop (Component Object Model) provides access to Word objects (sections, paragraphs, tables, etc.) and has functionality for creating and editing documents,

however, it does not support server-side automation and processes documents markedly slower than SDK.

The analysis of documents demanding certain formatting resulted in identification of a number of essential parameters for assessing the accuracy of text document formatting. Thus, the system is to perform compliance assessment according to these parameters:

- 1) page layout (page margins, paper format, orientation, columns, page numbers, header/footer settings);
- 2) paragraph (spacing, indentation, alignment);
- 3) font (size, name, color, toggle properties – bold, italics, underlined);
- 4) numbering and lists (level, numbering format, start value);
- 5) tables (vertical and horizontal text alignment, borders, cell margins, width, table header);
- 6) images (placement – anchor or inline, size).

The process of text documents check can be divided into several stages. Firstly, it is needed to define a set of rules according to which compliance assessment will be performed. The system provides a user with a possibility to specify design requirements for various documents by loading a formatting template or entering parameters manually, modify these requirements, or delete them if necessary; all information is stored in a formatting rules repository.

The second step is to upload the document into the system and select appropriate formatting rules. After that check of a document can be performed. The system reloads the document and adds comments with identified inconsistencies between the formatting used in a checked document and specified formatting requirements (see Fig. 3). So, in this case the system has detected that sizes of a header and a footer, page margin sizes, and some settings of a style "Heading 1" were selected incorrectly, and all this information was reported to the user. It should be noted that if there are no formatting mistakes in the original document, the system will not create any annotations. Comments on inaccurate paragraphs styling are added accordingly to each paragraph with incorrect formatting; notes on violation of formatting requirements for page layout settings, header/footer, etc. are added to the first paragraph of text. If there are formatting errors inside paragraphs or runs of text (for instance, some word has odd font settings), the system makes comments on each word in particular.

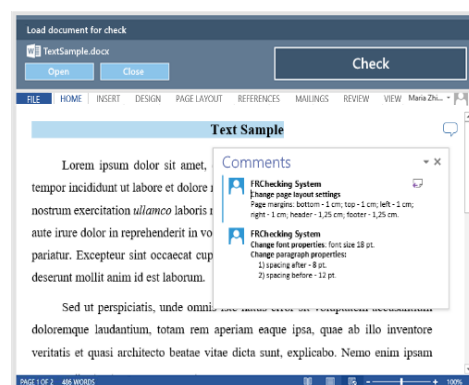


Fig. 3. Compliance assessment system interface

Thus, the user-system interaction complies with a number of different scenarios. The first scenario (see Fig. 4) implies that a user enters formatting rules manually, and then loads an original document for the check. In this case, the system (FRC System – Formatting Rules Check System) provides a user with either the resulting document containing the notes or the one with formatting corrected in accordance with rules specified by a user.

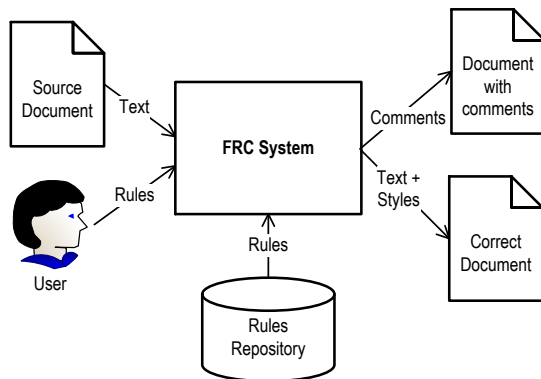


Fig. 4. Correct document generation

According to the second scenario (see Fig. 5) a user uploads a properly formatted template document, the system performs its analysis and downloads its formatting rules into the rules repository. This procedure significantly simplifies the entry of formatting rules of a document.

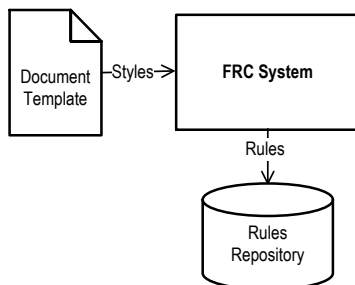


Fig. 5. Creation of rules based on document template

The third scenario of the interaction (see Fig. 6) suggests that a user manually enters formatting rules of a document, the system saves them in the repository, and then generates a document template with an automatically created styles which a user can use for further work with a document.

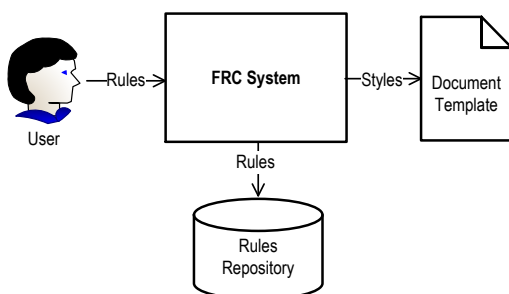


Fig. 6. Creation of document template based on rules

## V. TEXT DOCUMENT STRUCTURE CHECK

As noted earlier, the task of a text document analysis is not reduced to formatting rules check. In the more general case, it is necessary to analyze document structure, i.e. verify that all required sections are included. This problem often arises in preparation of design documentation, for example, in the process of developing information systems. Design documentation has a normative function, i.e. it contains mutual obligations of participants of a project that helps to avoid misunderstandings and abuses at the stage of handover-acceptance [10; 11].

The types and completeness of project documents are standardized. However, due to the fact that all technical documents are structurally very similar (they all consist of sections and subsections, may include additional documents, diagrams, tables, etc.), a special language for defining document structure and links between different documents can be developed. It will allow automating the process of analysis of an original set of project documents and generation of the new ones [12]. In the same way, it is reasonable to develop tools for extracting system requirements from the project documentation, and then control their compliance in the process of implementing the system. However, the process of creating design documents is quite a laborious task that requires precise knowledge of a document structure. This process can also be automated. Means of automating the generation of project documentation will allow generating a document template on the basis of descriptions of different sections of a document specified in a convenient visual user interface. This template can later be modified manually.

In order to describe documentation used in the process of information systems design, visual domain-specific language can be developed. Domain-Specific Language (DSL) is a modeling language designed for solving problems of a certain class in a particular domain. Unlike general-purpose modeling languages, DSL is more expressive, easy to use and intelligible to various categories of professionals, since it operates with the familiar terminology of the domain. Therefore, a large number of DSLs is designed nowadays in order to describe systems in different subject areas: artificial intelligence systems, distributed systems, mobile applications, real-time and embedded systems, simulation systems, etc.

Since description of project documents implies not only determining their structure, but also specifying the relations between them, the developed domain-specific language describing project documentation has two levels [13].

The first level of the language makes it possible to describe a set of documents and relations between them, the second level – the structure of a particular document. Due to a simple graphical notation of the language, the system can be used by IT-specialists, as well as clients who are not professional programmers.

## VI. CONCLUSION

The main result of the work done is the developed system that automates the check of a text document in accordance with formatting rules specified by a user. As it was tested, the

system substantially reduces the complexity of operations performed and makes the process less time-consuming.

Moreover, the visual DSL for describing the structure of a document was created. This language can be integrated into the support system of work of an analyst when information systems are designed. On the one hand, this provides means to perform analysis and parsing of a set of design documents loaded into system, presenting the sections of a document as individual elements of a model. On the other hand, with the use of the developed language an analyst can describe each section of a design document separately, and then generate a single text description on their basis.

Despite the fact that the system performs all the main functions, there is still space for improvement. The system can be upgraded by developing web-interface for more convenient use and expanding the set of criteria for document check in order to perform more comprehensive compliance assessment.

#### REFERENCES

- [1] S. Berdachuk, *Use the DocBook for Documentation Writing*. [[http://www.berdaflex.com/ru/eclipse/books/rcp\\_filemanager/ch01s04.html](http://www.berdaflex.com/ru/eclipse/books/rcp_filemanager/ch01s04.html)] (Checked: 10.04.2016).
- [2] S.M. Lvovsky, *Typing and formatting in LaTeX System*. Moscow: MTSNMO, 2006.
- [3] Orfogrammka. Spelling Checking Web Service. [<http://orfogrammka.ru>] (Checked: 10.04.2016).
- [4] A.A. Sokolov, A.M. Dvoryankin, A.Yu. Uzhva, "Development of the Method of Process of Technical Documentation Normative Control Automation," in *Izvestia VSTU*, 2013, no. 22 (125), pp. 116-117.
- [5] K.E. Klementyev, *Internal MS WORD document format* [<http://uinc.ru/articles/39/>] (Checked: 10.04.2016).
- [6] ISO/IEC 29500. Information technology – Document description and processing languages – Office Open XML File Formats. International Organization for Standardization, Geneva, Switzerland, 2012.
- [7] Microsoft. Microsoft by the Numbers [<http://news.microsoft.com/bythenumbers/planet-office/>] (Checked: 10.04.2016).
- [8] OpenXMLDeveloper.org. Benefits of Open XML. [<http://openxmldeveloper.org/wiki/w/wiki/benefits-of-open-xml.aspx>] (Checked: 21.10.2015).
- [9] W. Vugt, *Open XML Explained*. [[http://openxmldeveloper.org/cfs-file.ashx/\\_\\_key/communityserver-components-postattachments/00-00-00-19-70/Open-XML-Explained.pdf](http://openxmldeveloper.org/cfs-file.ashx/__key/communityserver-components-postattachments/00-00-00-19-70/Open-XML-Explained.pdf)] (Checked: 21.10.2015).
- [10] A.V. Zaboileeva-Zotova, Yu.A. Orlova, "Automation of Procedures of the Product Requirements Document Text Semantic Analysis," in *Izvestia VSTU*, 2007, no 3, vol. 9, pp. 52-55.
- [11] Yu.A. Orlova, "Product Requirements Document Text Analysis Methods," in *Izvestiya TSU. Engineering Sciences*, 2011, no 3, pp. 213-220.
- [12] M.A. Zhigalova, A.O. Sukhov, "Validation of the Design Documentation Based on Domain-specific Language," in *Vestnik molodykh uchenykh PSNRU*. Vol. 4. P. 224-228.
- [13] M.A. Zhigalova, A.O. Sukhov, "Domain-specific Language for Describing Documents Used in Information Systems Design," in *Izvestiya SFedU. Engineering Sciences*, 2015, no. 2, pp. 126-134.

# Complete contracts through specification drivers

Alexandr Naumchev\*, Bertrand Meyer†

Software Engineering Laboratory

Innopolis University

Innopolis, Russian Federation

†Also Politecnico di Milano

Email: \*a.naumchev@innopolis.ru, †Bertrand.Meyer@inf.ethz.ch

**Abstract**—Existing techniques of Design by Contract do not allow software developers to specify complete contracts in many cases. Incomplete contracts leave room for malicious implementations. This article complements Design by Contract with a simple yet powerful technique that removes the problem without adding syntactical mechanisms. The proposed technique makes it possible not only to derive complete contracts, but also to rigorously check and improve completeness of existing contracts without instrumenting them.

**Index Terms**—specification driver, abstract data type, Design by Contract, complete contract, Eiffel, Hoare triple, AutoProof

## I. INTRODUCTION

The main contribution of this work is a new approach to seamless software development, bridging the heretofore wide gap between two fundamental and widely used techniques: Abstract Data Types (ADTs) and Object-Oriented Programming (OOP). These techniques seem made for each other, but trying to combine them in practice reveals a glaring impedance mismatch. We explain the problem, provide a remedy, and subject it to formal verification.

ADTs [1] are a clear, widely known way to specify systems precisely. OOP [2] is the realization of ADT ideas at the design and programming level, with Design by Contract (semantic properties embedded in the program) providing the connection. At least, that is the accepted view. However, the correspondence is far less simple than this view would suggest. While it would seem natural to use ADTs for specification and OOP for design and implementation, in practice this combination hits an impedance mismatch:

- At the ADT level, some axioms involve two or more commands. For example, an axiom for stacks (the standard example of ADTs, which remains the best for explanatory purposes) will state that if you push an element onto a stack and then pop the stack, you end up with the original stack.
- In a class, the standard unit of OOP, the contracts can only talk about one command, such as push or pop, but not both. Specifically, the postcondition of a command such as push can describe the command's effect on queries such as top (after you have pushed an element, that element is the new top), but there is no way to refer to the effect on pop as expressed by the ADT axiom.

The present work introduces a practical solution to this mismatch. The essence of the solution is that classes directly

reflecting ADTs, such as a class STACK, cannot by themselves capture such multi-command (or "second-degree") ADT axioms, but this does not mean that the OOP approach fails us. The idea will be to introduce auxiliary classes whose role is to "talk about" the features of the basic classes such as STACK (the ones directly corresponding to ADTs). Such a class has features that combine those of basic classes, e.g. a command `push_then_pop` that works on an arbitrary stack, pushing an element on a stack and then popping the stack. Then the postcondition of `push_then_pop` can specify that the resulting stack is the same as the original.

We call such features specification drivers by analogy with "test drivers", which are similarly added to the basic units of a system for the sole purpose of testing them. Like test drivers, specification drivers serve purely verification purposes, rather than providing system functionality. The difference is of course that test drivers appear in dynamic verification (testing), whereas specification drivers are for static verification (for example, as in this paper, correctness proofs). But the basic idea is the same.

Specification drivers are not just a specification technique; we also submit them to formal, mechanical verification. As part of the AutoProof formal verification tool [3], we have mechanically proved the correctness of the examples given in this paper.

Section II explains the problem through a working example. Section IV describes the essentials of the solution. Section V compares this approach with other possible ones. Section VI presents our experience with mechanical verification. Section VII draws conclusions and outlines future research prospects.

## II. MOTIVATING EXAMPLE

Figure 1 contains the standard ADT specification of stacks. The standard names of the functions are changed in favor of the mechanical verification experiment in Section VI: the existing implementation, to which the experiment is applied, uses exactly these names.

Figure 2 contains the result of applying the traditional process of DbC [2] to the specification in Figure 1:

- The name of the class is derived from the name of the ADT it implements.
- The signatures of the implementation features are derivatives of the ADT functions' descriptions.



- Preconditions of the ADT functions go to **require** clauses of the implementation features.
- Postconditions of the implementation features capture ADT axioms A1, A3 and A4.
- The **create** clause lists the implementation feature *new* to highlight its special mission of instantiating new stacks.

Axiom A2 introduces the problem. The axiom constrains

## TYPES

- $STACK[G]$

## FUNCTIONS

- $extend : STACK[G] \times G \rightarrow STACK[G]$
- $remove : STACK[G] \rightarrow STACK[G]$
- $item : STACK[G] \rightarrow G$
- $is\_empty : STACK[G] \rightarrow BOOLEAN$
- $new : STACK[G]$

## AXIOMS

For any  $x : G, s : STACK[G]$

- (A1)  $item(extend(s, x)) = x$
- (A2)  $remove(extend(s, x)) = s$
- (A3)  $is\_empty(new)$
- (A4) **not**  $is\_empty(extend(s, x))$

## PRECONDITIONS

- (P1)  $remove(s : STACK[G])$  **require not**  $is\_empty(s)$
- (P2)  $item(s : STACK[G])$  **require not**  $is\_empty(s)$

Fig. 1. ADT specification of stacks

```
class STACK_IMPLEMENTATION [G] -- Type STACK[G]
create new -- Marking new as a creation feature
feature
  extend (x: G) -- Extending with a new element
  do
    ensure
      a1: item = x
      a4: not is_empty
    end

  remove -- Removing the topmost element
  require
    p1: not is_empty
  do
  end

  item: G -- The topmost element
  require
    p2: not is_empty
  do
  end

  is_empty: BOOLEAN -- Is the stack empty?

  new -- Instantiating a stack
  do
    ensure
      a3: is_empty
    end
  end
end
```

Fig. 2. Applying the traditional process of DbC to the stacks ADT specification

```
class STACK_IMPLEMENTATION [G]
create new
feature
  extend (x: G) -- Extending with a new element
  do
    item := x
    is_empty := False
  ensure
    a1: item = x
    a4: not is_empty
  end

  remove -- Removing the topmost element
  do
    is_empty := True
  end

  item: G -- The topmost element

  is_empty: BOOLEAN -- Is the stack empty?

  new -- Instantiating a stack
  do
    is_empty := True
  ensure
    a3: is_empty
  end
end
```

Fig. 3. Underspecified postconditions may lead to invalid implementations

two functions simultaneously, *extend* and *remove*: the former one should do nothing but extend the stack with the given element, and the latter should do nothing but remove the topmost element of the stack. As a consequence, it is not possible to capture the axiom in a single implementation feature postcondition. Postconditions operate on two objects: the target object before calling the feature and the target object after invoking the feature. If the feature has formal parameters, they also parameterize the postcondition. Axiom A2 involves three stacks: the original one  $s$ ,  $s_1$  resulting from applying function *extend* to  $s$ , and finally  $s_2$  resulting from applying *remove* to  $s_1$ . Formally:

$$\forall s, s_1, s_2 : STACK[G]; x : G \bullet \\ (s_1 = extend(s, x) \wedge s_2 = remove(s_1) \Rightarrow s_2 = s)$$

Or, writing the quantified expression in terms of postconditions:

$$(Post_{extend}(s, s_1, x) \wedge Post_{remove}(s_1, s_2)) \Rightarrow s_2 = s \quad (1)$$

On one hand, it is not possible to capture A2 in a single postcondition. On the other hand, postconditions of *extend* and *remove* should exist and be strong enough to satisfy Equation 1.

Failures to capture such important properties as A2 in postconditions leave room for invalid implementations. In particular, inability to capture axiom A2 makes it possible to implement stacks which store only the last added element and thus are useless as data containers. Still, such an implementation satisfies all the other axioms as its postconditions capture them.



Figure 3 depicts such an invalid implementation. For the sake of simplicity, it ignores preconditions, but this does not render the reasoning invalid: an empty precondition defaults to `TRUE`, the weakest conceivable precondition. According to the rule of consequence for preconditions [4], correctness against a weaker precondition implies correctness against a stronger one. Submitting the class `STACK_IMPLEMENTATION` to AutoProof confirms the point: the tool successfully proves "correctness" of the implementation.

For purist developers the problem of underspecified postconditions may easily become a reason for not using them at all. Intuitively, it seems better to keep all the properties written in a single place, and the described problem prevents doing this: although it is possible to capture some ADT axioms in postconditions, some of them will have to exist in separate documents and thus carry the risk of misuse and all the associated traceability costs.

### III. AXIOMS AS SPECIFICATION DRIVERS

The example in Figure 2 translates axiom A1 directly to the postcondition of the implementation feature `extend`. Is it in fact the only way to do the translation of the axiom? A closer look at the original axiom and its translation in Figure 2 reveals two facts:

- The axiom uses the function *extend* in a sense of applying it, while its translation in Figure 2 specifies the implementation feature directly without invoking it.
- The axiom uses an explicit stack instance *s*, while the translation implicitly operates on the current object described by class `STACK_IMPLEMENTATION[G]`.

Is it possible to devise a translation of axiom A1 that would be closer to the origin?

Existing techniques of DbC completely ignore a large family of program constructs: features with pre- and postconditions whose only purpose is to serve as proof obligations. Such features do not implement any ADT functions and are not to be invoked. Instead, they are intended solely for static verification.

Figure 4 gives an example. The feature `extend_updates_item` is an alternative translation of axiom A1. It possesses the following properties:

- It operates on explicit objects *s* and *x*.
- It uses an explicit invocation of implementation feature `extend`.

The example in Figure 4 takes the whole feature `extend_updates_item` as the translation of the axiom, as opposed

```
extend_updates_item (s: STACK_IMPLEMENTATION [G]; x: G)
do
  s.extend(x)
ensure
  s.item = x
end
```

Fig. 4. Axiom A1 as a specified feature

```
remove_then_extend (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
  require
    s1.is_equal(s2)
  do
    s1.extend (x)
    s1.remove
  ensure
    s1.is_equal(s2)
end
```

Fig. 5. Axiom A2 as a specified feature

to the one in Figure 2, where the axiom is captured with the assertion `item = x` in the postcondition of implementation feature `extend`.

Using this approach, it is possible to capture axiom A2 in the form of the feature `remove_then_extend` in Figure 5. Again, the whole feature is the translation of the axiom. The feature `is_equal` defines an equivalence relation over run time objects representing stacks. It is declared by default in all Eiffel classes and compares its operands by value. The notion of equality deserves a separate analysis; Section IV-B gives the details.

Henceforth, this article will use the term **specification drivers** for specified features serving as translations of certain ADT axioms. A specification driver can be proven correct only if the implementation features it invokes have strong enough postconditions. Consequently, specification drivers, as their name suggests, drive specifying stronger postconditions.

### IV. SPECIFICATION DRIVERS IN PRACTICE

The present section derives the complete set of specification drivers for the stacks ADT (Figure 1). This set includes not only specification drivers that directly represent the original axioms of stacks because some specification drivers stem from a fundamental difference between ADT specifications and object-oriented programs: in the former it is not possible to have more than one occurrence of one and the same abstract stack, while in the latter it is possible to instantiate two run time objects denoting one and the same abstract stack. Section IV-B and Section IV-C discuss the issue in detail and derive additional specification drivers caused by it.

#### A. ADT axioms

Specification drivers do not bring any functional value to the system: they exist only to be eventually discharged as proof obligations. Consequently, they should not pollute implementation classes like `STACK_IMPLEMENTATION` in Figure 2. Concerning where to store them, the simplest option is to create a separate class within the source code project. The `ADT_AXIOMS_SPECIFICATION_DRIVERS` class in Figure 6 contains specification drivers capturing the ADT axioms of stacks. This class is generic: since it talks about instances of a generic concept, `STACK_IMPLEMENTATION [G]` in this case, it needs to assume existence of type *G* to keep the genericity. The `{NONE}` clause suggests that the features listed within the corresponding **feature** block do not supply any useful functionality. The `deferred` keyword in front of the class declaration suggests that

it is not possible to instantiate any objects of this class, which makes sense as the class serves as a document containing specification drivers rather than a blueprint for creating run time objects.

### B. Equivalence

It is possible to see that the specification drivers in Figure 6 use two different operators for objects comparison: `=` and `is_equal`, while the original ADT specification in Figure 1 invokes only `=`. This section discusses the difference between comparing instances of ADTs and comparing objects instantiated from object-oriented classes and introduces a set of specification drivers capturing the difference.

ADT specifications operate on sets of instances in the mathematical sense of the word "set": an abstract data type cannot contain two instances of one and the same abstract object. For example, the range of the function *new* consists of the only stack instance, which is the empty stack, as axiom A4 suggests. When an object-oriented program is running, it is perfectly fine for it to have two run time objects in its memory denoting one and the same instance of the ADT. For example, it is possible to declare two variables of type `STACK_IMPLEMENTATION [INTEGER]` and make them both refer to two different stack objects in the memory, as in Figure 7. Consequently, run time objects form not a set of abstract objects, but a *multiset*, or *bag* [5]. That is why there are two different comparison operators: the `=` operator checks whether

```
s1, s2: STACK_IMPLEMENTATION [INTEGER]
create s1.new
create s2.new
```

Fig. 7. Creating two instances of the empty stack

```
deferred class EQUIVALENCE_SPECIFICATION_DRIVERS [G]
feature {NONE}
  reflexivity (s: STACK_IMPLEMENTATION [G])
  do
    ensure
      s.is_equal (s)
  end

  symmetry (s1, s2: STACK_IMPLEMENTATION [G])
  require
    s1.is_equal (s2)
  do
    ensure
      s2.is_equal (s1)
  end

  transitivity (s1, s2, s3: STACK_IMPLEMENTATION [G])
  require
    s1.is_equal (s2)
    s2.is_equal (s3)
  do
    ensure
      s1.is_equal (s3)
  end
end
```

Fig. 8. Capturing the definition of equivalence

```
deferred class ADT_AXIOMS_SPECIFICATION_DRIVERS [G]
feature {NONE}
  axiom_a1 (s: STACK_IMPLEMENTATION [G]; x: G)
  do
    s.extend (x)
  ensure
    s.item = x
  end

  axiom_a2 (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
  require
    s1.is_equal (s2)
  do
    s1.extend (x)
    s1.remove
  ensure
    s1.is_equal (s2)
  end

  axiom_a3 (s: STACK_IMPLEMENTATION [G]; x: G)
  do
    s.extend (x)
  ensure
    not s.is_empty
  end

  axiom_a4: STACK_IMPLEMENTATION [G]
  do
    create Result.new
  ensure
    Result.is_empty
  end
end
```

Fig. 6. Specification drivers capturing the axioms of stacks

the operands refer to identical run time objects, and `is_equal` checks whether the objects referenced by the operands represent the same instance of the ADT implemented by the class. As a consequence, if specification drivers representing ADT axioms use the feature `is_equal`, the corresponding implementation class should redefine the feature and its postcondition should be strong enough to satisfy the definition of equivalence relations. A relation over stacks is an equivalence relation if and only if it possesses the following properties:

- Reflexivity: every stack is equal to itself.
- Symmetry: if stack  $s_1$  is equal to stack  $s_2$ , then  $s_2$  is equal to  $s_1$  as well.
- Transitivity: if stack  $s_1$  is equal to stack  $s_2$ , and  $s_2$  is equal to  $s_3$ , then  $s_1$  is equal to  $s_3$ .

As Figure 8 illustrates, the three properties may be captured by a separate class created specifically for this goal. If all the features of class `EQUIVALENCE_SPECIFICATION_DRIVERS` are correct, then the postcondition of `is_equal` indeed defines an equivalence relation over run time objects instantiated from `STACK_IMPLEMENTATION [G]`.

It is worth noting that because equivalence definition is static, specification drivers for equivalence may be generated automatically for every class.

### C. Well-definedness

The ADT specification in Figure 1 lists certain functions over stacks. It is necessary to ensure that they remain func-

```

deferred class WELL_DEFINEDNESS_SPECIFICATION_DRIVERS [G]
feature {NONE}
  new_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    s1.is_empty
    s2.is_empty
  do
  ensure
    s1.is_equal (s2)
  end

  is_empty_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    s1.is_equal (s2)
  do
  ensure
    s1.is_empty = s2.is_empty
  end

  item_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    not s1.is_empty
    not s2.is_empty
    s1.is_equal (s2)
  do
  ensure
    s1.item = s2.item
  end

  extend_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G]; x: G)
  require
    s1.is_equal (s2)
  do
    s1.extend (x)
    s2.extend (x)
  ensure
    s1.is_equal (s2)
  end

  remove_is_well_defined (s1, s2: STACK_IMPLEMENTATION [G])
  require
    not s1.is_empty
    not s2.is_empty
    s1.is_equal (s2)
    s1 ≠ s2
  do
    s1.remove
    s2.remove
  ensure
    s1.is_equal (s2)
  end
end

```

Fig. 9. Specification drivers for well-definedness

tions in the presence of an equivalence relation. Invoking a given implementation feature for two run time objects, which represent a single ADT object, should be indistinguishable from applying the ADT function implemented by this feature to that ADT object. Since a function application produces only one element from its range set, the two run time objects should also be considered equal after the invocation. This property is called **well-definedness** under an equivalence relation [6]. The class `WELL_DEFINEDNESS_SPECIFICATION_DRIVERS` in Figure 9 contains specification drivers that encode well-definedness for every stacks implementation feature. The specification drivers `item_is_well_defined` and `remove_is_well_defined`

contain assertions `not s1.is_empty` and `not s2.is_empty`. These specification drivers invoke implementation features `item` and `remove`, which have preconditions that need to be satisfied. The purpose of the mentioned assertions is exactly this. The  $s1 \neq s2$  assertion in the precondition of the specification driver `remove_is_well_defined` is there for a very specific reason. If  $s1$  and  $s2$  are identical, the precondition for the `s2.remove` call may not hold: even if the stack object referenced by  $s1$  and  $s2$  is not empty in the beginning, it may not be the case anymore after the `s1.remove` call. This additional assertion does not remove any generality: indeed, identity always implies equality, and proving the latter is exactly the purpose of this specification driver, according to its postcondition.

Specification driver `new_is_well_defined` deserves special attention too. In fact, it encodes something stronger than just the well-definedness of the implementation feature `new`. It says that two empty stacks are always equal. This makes perfect sense and at the same time implies the necessary well-definedness property: from the ADT specification in Figure 1 and its first approximation in Figure 2, it is known that instantiating a stack with function `new` results in the empty abstract stack. Consequently, the `new_is_well_defined` specification driver covers this case, since it applies to every pair of run time objects denoting the empty abstract stack.

Similarly to equivalence, the notion of well-definedness is long-established; as such, it may be possible to generate the corresponding specification drivers automatically.

#### D. Complete contracts

Although some works ([7], [8]) talk about contract (in)completeness, they do not define this notion precisely. In light of the fundamental difference between ADT specifications and object-oriented programs, which causes the notion of equivalence over run time objects to appear (Section IV-B), the definition cannot be implicitly equal to the definition of sufficiently complete ADT specifications [9] and needs to be written down explicitly.

As the other details of the original definition in [2] do not bring any value to the discussion, this article uses a simplified definition of a contract.

**Definition 4.1:** A **contract** is a set composed of all pairs of the form  $(Precondition(f), Postcondition(f))$  for every implementation feature  $f$ .

This definition ignores the possible presence of class invariants as it is always possible to get rid of them by appending to pre- and postconditions of the implementation features.

**Definition 4.2:** A contract is **correct** if and only if:

- Its postconditions are strong enough to ensure correctness of the specification drivers derived from the input ADT axioms (Section IV-A)
- In the event that specification drivers for the input ADT axioms use equivalence, its postconditions are strong enough to ensure correctness of the specification drivers for equivalence (Section IV-B).

*Definition 4.3:* A contract is **well-defined** if and only if its postconditions are strong enough to ensure correctness of the specification drivers for well-definedness (Section IV-C).

*Definition 4.4:* A contract is **complete** if and only if it is correct and well-defined.

## V. RELATED WORK

Doctoral thesis [7] uses features with pre- and postconditions for checking completeness of model-based contracts (discussed later in this section). The definition of a complete model-based contract is not related to the definition of completeness in Section IV-D. According to [7], completeness is what the current article calls well-definedness, expressed in terms of abstract mathematical concepts.

Although the specification driver approach allows capturing ADT axioms in their original form, it does specify how to actually build complete contracts having a set of specification drivers. As Section II suggests, in many cases it is not possible to specify strong enough postconditions in terms of the ADT specification itself. This is where the need for representation appears: the implementation class has to stick to some already implemented data structure in order to enable stronger postconditions expressed in terms of this data structure. The problem of choosing an ideal representation has been aptly handled in multiple publications, therefore the present article does not propose its own methodology, but chooses instead to reference these publications.

Work [10] shows that it makes sense to use mathematical abstractions for representations: for example, it seems reasonable to think about stacks as mathematical sequences. That work also shows how to prove correctness against contracts strengthened with precise mathematical abstractions. Work [8] introduces the Mathematical Model Library (MML) - Eiffel library containing core abstractions: sets, sequences, bags, tuples etc. A more recent work [11] introduces EiffelBase2, a usable library of essential data structures, including stacks, represented as mathematical abstractions from MML. EiffelBase2 is fully verified with the AutoProof verifier. The underlying verification methodology [12] assumes writing quite a number of assertions related to program execution semantics, so giving complete examples here would introduce confusion rather than clarity. Instead, Figure 10 presents the idea in a nutshell. The `STACK_SEQUENCE_IMPLEMENTATION` class is the abstract model of stacks from the EiffelBase2 standpoint. EiffelBase2 equips classes implementing stacks with the `sequence` attribute and strengthens postconditions of the implementation features in terms of it. Class `MML_SEQUENCE` cannot be instantiated into any run time objects and exists only for verification purposes: it maps directly to the data structure representing mathematical sequences in the underlying proving engine. The `sequence` attribute is further connected to meaningful data structures by means of abstraction and refinement techniques [13]. Works [11] and [7] give more implementation details.

In Figure 10, the implementation features are formally defined with assertions over the `sequence` attribute (marked with the "definition" tag) added to the features' postconditions. The

```
class STACK_SEQUENCE_IMPLEMENTATION [G]
inherit ANY redefine is_equal end
create new -- Marking new as a creation feature
feature
  sequence: MML_SEQUENCE [G] -- Stack representation

  extend (x: G) -- Extending with a new element
  do
    ensure
      a1: item = x
      a4: not is_empty
      definition: sequence = old sequence.extended (x)
    end

  remove -- Removing the topmost element
  require
    not is_empty
  do
    ensure
      definition: sequence = old sequence.but_last
    end

  item: G -- Retrieving the topmost element
  require
    not is_empty
  do
    ensure
      definition: Result = sequence.last
    end

  is_empty: BOOLEAN -- Is the stack empty?
  do
    ensure
      definition: Result = sequence.is_empty
    end

  new -- Instantiating a stack
  do
    ensure
      a3: is_empty
      definition: sequence.is_empty
    end

  is_equal (other: STACK_SEQUENCE_IMPLEMENTATION [G]): BOOLEAN --
    Redefining equality
  do
    ensure then
      definition: Result = (sequence.count = other.sequence.count
        and then
          (across 1 .. sequence.count as i all sequence[i.item] =
            other.sequence[i.item] end))
    end
  end
end
```

Fig. 10. Abstract model of stacks as sequences

comparison feature `is_equal` is redefined so that two stacks are considered equal if and only if the sequences representing them are equal. Two sequences are considered equal if and only if their sizes are equal and they contain same objects. The feature `extended` models a sequence where an object is appended to the target sequence on to which the feature is invoked; feature `but_last` models the target sequence, but without the last element; feature `last` models the element added to the target sequence last; feature `is_empty` models the indication whether the target sequence is empty or not; finally, feature `count` models the size of the target sequence.

Mathematical concepts from MML are abstract, but they still form particular representations in EiffelBase2, though mathematically precise. The concept of model-based contracts helps to specify complete contracts, but does not say how to rigorously check contracts with representations for completeness. Furthermore, it fails to define what complete contracts are. The notion of specification drivers bridges this gap. All the specification drivers derived in the present article are expressed in terms of the original ADT specification (Section IV-A) plus the abstract equivalence (Section IV-B and Section IV-C), whose presence is inevitable due to the nature of computing which allows programs to keep in their memory several instances of one and the same abstract object. They do not require making any assumptions about possible representations and enable defining complete contracts precisely.

## VI. PROVING CONTRACTS COMPLETENESS

It is possible to give a manual proof of completeness of the contract depicted in Figure 10. Fortunately, this work may be done automatically. This advantage makes it possible to apply the specification drivers approach to legacy implementations. Indeed: if there is a source code project with a number of classes in it, then it is possible to devise an additional class, write all the applicable specification drivers into it and submit the resulting class to the prover. Instead of showing how to derive complete contracts having a set of specification drivers from scratch, the article shows how to apply the approach to existing contracts.

The EiffelBase2 library seems to be a natural choice for the experiment. The library contains a complete implementation of stacks specified as mathematical sequences. The corresponding implementation class is `V_LINKED_STACK`. In order to perform the experiment, it is necessary to take the stacks specification drivers from Section IV and modify them so that the name of the implementation class would be `V_LINKED_STACK` instead of `STACKS_IMPLEMENTATION`. The specification driver `axiom_a4` comes with a pitfall: the `V_LINKED_STACK` class does not introduce its own creation feature, but redefines the default creation feature defined for all classes. Hence, the `create Result.new` instruction is not applicable here; one should use `create Result` instead. After these modifications, the specification drivers should successfully compile and be ready for verification.

The initial verification attempt using AutoProof will result in numerous precondition violations. As Section V suggests, the verification methodology [12] behind AutoProof assumes writing additional non-stack related assertions. For example, the `extend_is_well_defined` specification driver can be verified by AutoProof only in the form depicted in Figure 11. The five assertions in the beginning of the `require` precondition clause seem to be worth explaining them briefly. The `s1.is_wrapped` assertion says that reference `s1` is assumed to be non-void and not participating in any call; the `s1.observers.is_empty` assertion says that the set of objects interested in the state of `s1` should be empty - it is a part of the precondition of feature `extend` of class `V_LINKED_STACK`; finally, the `modify([s1, s2])` assertion is a frame specification: it says that the enclosing feature,

```

extend_is_well_defined (s1, s2: V_LINKED_STACK [G]; x: G)
  require
    s1.is_wrapped
    s2.is_wrapped
    s1.observers.is_empty
    s2.observers.is_empty
    modify([s1, s2])

    s1.is_equal (s2)
  do
    s1.extend (x)
    s2.extend (x)
  ensure
    s1.is_equal (s2)
  end

```

Fig. 11. Specification driver for verifying by AutoProof

`extend_is_well_defined` in this case, is going to modify objects referenced by `s1` and `s2` (square brackets `[]` denote set constants in Eiffel). The precondition needs the `modify` assertion because the `extend_is_well_defined` feature uses feature invocations with side effects, `extend` in this case, on references `s1` and `s2`. Although the verification failures caused by the absence of these assertions do not bear any relation to stacks, they uncover certain weaknesses in the verification methodology: namely, the defaults do not seem sufficiently reasonable. For example, a violation of the `s1.is_wrapped` assertion would detect a callback situation, and callbacks are not so common as to assume them by default. The `observers.is_empty` requirement makes extending stack objects applicable only in situations when no other objects depend on their states. The `modify` frame specification may be generated automatically based on the presence of invocations with side effects in the implementation body.

After complementing the specification drivers with all necessary assertions related to verification methodology and rerunning AutoProof, it uncovers some stack-related issues. This is visible from the fact that this time the verification errors come from the postconditions. Namely, AutoProof fails to prove correctness of all the verification drivers from classes `EQUIVALENCE_SPECIFICATION_DRIVERS` and `WELL_DEFINEDNESS_SPECIFICATION_DRIVERS` as well as verification driver `axiom_a2` from the `ADT_AXIOMS_SPECIFICATION_DRIVERS` class. As all of these specification drivers involve implementation feature `is_equal`, the first guess is that `V_LINKED_STACK` does not redefine it. This guess appears to be right: the class defines its own custom feature for comparing run time objects, but does not redefine the standard comparison feature in terms of the new one. Giving this flaw's fix here would not bring much value to the discussion, so it seems better to move on. After redefining feature `is_equal`, AutoProof succeeds in proving classes `ADT_AXIOMS_SPECIFICATION_DRIVERS` and `EQUIVALENCE_SPECIFICATION_DRIVERS` completely, but still fails to prove specification driver `new_is_well_defined` from the `WELL_DEFINEDNESS_SPECIFICATION_DRIVERS` class. As this specification driver uses the `is_empty` implementation feature, it falls



under suspicion. Apparently, its postcondition does not have a clause corresponding to the definition clause in its abstract model in Figure 10. After fixing this flaw, everything verifies successfully, including the `V_LINKED_STACK` implementation class.

## VII. CONCLUSIONS AND FURTHER WORK

The article makes the following main contributions:

- Presents the specification driver approach for encoding ADT axioms, which are not possible to encode using traditional DbC techniques.
- Illustrates the process of axiomatizing abstract equivalence using the new approach.
- Introduces an exhaustive definition of contract completeness.
- Demonstrates how to apply completeness checks to legacy implementations.

The new approach allows adding, changing or removing ADT axioms at any given moment of the development process without necessarily modifying the implementation classes. Although specification drivers occupy separate classes completely disjoint from implementation classes, they are simultaneously expressed in terms of objects instantiated from the implementation classes. The result is a seamless integration of software axiomatization and implementation driven by automatic verification of functional correctness. Attempts to check specification drivers can uncover weak postconditions of implementation features. Once strengthened, these postconditions potentially yield firmer executable instructions.

In light of the presence of different kinds of specification drivers described in Section IV it seems feasible to propose the following changes to the Eiffel Verification Environment tool set:

- Develop a template for fast creation of classes intended to keep specification drivers.
- Automate generation of specification drivers for equivalence and well-definedness.
- Revise verification methodology underlying AutoProof: in essence, specification drivers are a new syntactical specification construct, which may potentially remove some particularly egregious verification challenges.

Work [14] introduces the notion of multirequirements, and work [15] illustrates how to apply this notion in practice. The underlying idea is that a separate item in a software requirements document should be expressed using several interwoven notations, e.g. natural language, graphical form and formal notation. For the formal notation, it was suggested to use a rather expressive programming language. The present paper talks about expressing ADT axioms in a programming language with pre- and postconditions. Since ADT specifications are one of the languages for expressing software requirements, it makes sense to revisit the original multirequirements approach to see how the idea of specification drivers could improve it.

The idea of specification drivers was inspired mostly by seminal works [13] and [2], and driven by the will to unify requirements and code seeded in the work [14].

## ACKNOWLEDGMENT

The authors would like to thank Innopolis University for supporting the Software Engineering Laboratory where the research resulted in this work is taking place.

Special thanks goes to Daniel Johnston from MSIT-SE Program at Innopolis University, who kindly agreed to proofread this paper line-by-line.

## REFERENCES

- [1] J. Guttag, "Abstract data types and the development of data structures," *Communications of the ACM*, vol. 20, no. 6, pp. 396–404, 1977.
- [2] B. Meyer, *Object-oriented software construction*. Prentice hall New York, 1988, vol. 2.
- [3] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, "Autoproof: Auto-active functional verification of object-oriented programs," *arXiv preprint arXiv:1501.03063*, 2015.
- [4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.
- [5] W. D. Blizard *et al.*, "Multiset theory," *Notre Dame Journal of formal logic*, vol. 30, no. 1, pp. 36–66, 1989.
- [6] D. S. Dummit and R. M. Foote, *Abstract algebra*. Prentice Hall Englewood Cliffs, 1991, vol. 1999.
- [7] N. Polikarpova, "Specified and verified reusable components," Ph.D. dissertation, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21939, 2014, 2014.
- [8] B. Schoeller, T. Widmer, and B. Meyer, "Making specifications complete through models," in *Architecting Systems with Trustworthy Components*. Springer, 2006, pp. 48–70.
- [9] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta informatica*, vol. 10, no. 1, pp. 27–52, 1978.
- [10] B. Meyer, "A framework for proving contract-equipped classes," in *Abstract State Machines 2003*. Springer, 2003, pp. 108–125.
- [11] N. Polikarpova, J. Tschannen, and C. A. Furia, "A fully verified container library," in *FM 2015: Formal Methods*. Springer, 2015, pp. 414–434.
- [12] N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer, "Flexible invariants through semantic collaboration," in *FM 2014: Formal Methods*. Springer, 2014, pp. 514–530.
- [13] C. A. R. Hoare, *Proof of correctness of data representations*. Springer, 2002.
- [14] B. Meyer, "Multirequirements," in *Modelling and Quality in Requirements Engineering (Martin Glinz Festschrift)*, N. Seyff and A. Koziolok, Eds. MV Wissenschaft, 2013.
- [15] A. Naumchev, B. Meyer, and V. Rivera, "Unifying requirements and code: an example," To appear in Ershov Informatics Conference, PSI, Kazan, Russia (LNCS), 2016.

# Usability of AutoProof: a case study of software verification

Mansur Khazeev<sup>\*</sup>, Victor Rivera<sup>†</sup>, Manuel Mazzara<sup>‡</sup> and Alexander Tchitchigin<sup>§</sup>  
Innopolis University, Software Engineering Lab.

Innopolis, Russia

Email: <sup>\*</sup>m.khazeev@innopolis.ru, <sup>†</sup>v.rivera@innopolis.ru, <sup>‡</sup>m.mazzara@innopolis.ru, <sup>§</sup>a.chichigin@innopolis.ru

**Abstract**—Many verification tools come out of academic projects, whose natural constraints do not typically lead to a strong focus on usability. For widespread use, however, usability is essential. Using a well-known benchmark, the Tokeneer problem, we evaluate the usability of a recent and promising verification tool: AutoProof. The results show the efficacy of the tool in verifying a real piece of software and automatically discharging nearly two thirds of verification conditions. At the same time, the case study shows the demand for improved documentation and emphasizes the need for improvement in the tool itself and in the Eiffel IDE.

## I. INTRODUCTION

Modern systems have grown fast in complexity, and demand for quality. Focus on quality, in turn, demands stronger attention to the entire development life-cycle. With the tendency to reuse and integration, the need for software quality is even more important since applications and components have to rely on each other with partial knowledge of the implementation and based on interface only.

Tools for software verification allow the application of theoretical principles in practice, in order to ensure that nothing bad will ever happen (safety). The extra effort required by the use of these tools is certainly not for free and comes with increased development costs [1]. There is a common belief in industry that developing software with high level of assurance is too expensive, therefore not acceptable, especially for non safety-critical or financially-critical applications.

Tools and techniques for the formal development of software have played a key role on demystifying this belief. There are several approaches, for instances abstract interpretation and model checking [2], [3], that seek the automation to formally proving certain conditions of systems. However, these techniques tend to verify simple properties only. On the other end of the spectrum, there are interactive techniques for verification such theorem provers [4]. These techniques aim at more complex properties, but demand the interaction of users to help the verification.

Nowadays, there are new approaches that aim at finding a good trade-off between both techniques, e.g. auto-active: users are not needed during the verification process (it is automatically performed); they are required instead to provide guidance to the proof using annotations. AutoProof [5], is a static auto-active verifier for functional properties of object-oriented programs. Using AutoProof, users write code and

equip classes with contracts and annotations to help the tool to prove certain properties.

The main goal resented in this paper is to provide insights on how easy/difficult is for users (mainly engineers without deep knowledge of formal verification) to use current methodologies and tools for the development of software with high level of assurance, in particular on the use of the AutoProof tool.

Generally, to prove the correctness of a program one needs some mechanisms to express what the program is supposed to do and clearly state it in the specifications that are used later to verify the program. Eiffel programming language natively supports these mechanisms by means of contracts. Eiffel is an object-oriented programming language which directly implements the concepts of Design-by-Contract (DbC) [1], [6]. The key concept is viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations. This is realized equipping methods with pre- and post-conditions, and classes with invariants. The key feature of the Eiffel language is indeed the idea that all the methods might and should contain contracts.

Contracts and annotations used in Eiffel are used by AutoProof to statically verify the consistency of the classes. To demonstrate the usability of the tool, the Tokeneer project [7] was implemented in Eiffel and AutoProof was used to verify the consistency of the code. The Tokeneer project is a system specified and implemented by National Security Agency (NSA). Initially, NSA carried out this challenge to prove that it is possible to develop secure systems rigorously in a cost effective manner. Since its development, it became a testing range for different software development methodologies and verification tools. Results of the project are publicly available. This paper reports on the use of AutoProof to verify an Eiffel implementation of Tokeneer and also reports on how easy/difficult is for users to use the tool, e.g. the burden of helping the tool by means of annotations in the code.

The rest of the paper is organized as follows: Section II introduces the Tokeneer project, Eiffel and the AutoProof tool. Section III describes the methodology used to verify the implementation of the Tokeneer project. Section IV presents empirical results helping to draw conclusions. Section V is devoted to related work and Section VI concludes and mentions future work.



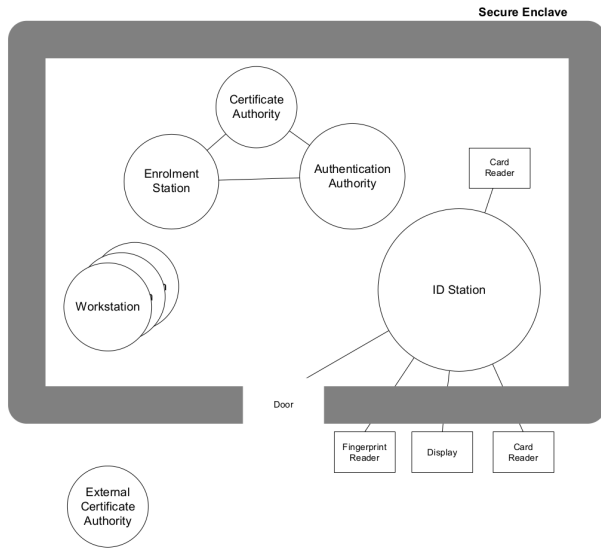


Fig. 1. The Tokeneer System

## II. PRELIMINARIES

### A. The Tokeneer Project

In 2002, with the aim to prove/disprove the common believe in industry that development of software of high level of assurance is too expensive and therefore not feasible, the National Security Agency (NSA) asked Altran to undertake a research project to develop part of an existing secure system, the Tokeneer System, in accordance with Altran's Correctness by Construction development process. The system was specified using Z notation [8] and implemented in Ada [9]. The project was successfully delivered in 2003 within 260 days of effort, and later, in 2008, all the results were made available by NSA to the software development and security communities in order to demonstrate the possibility to develop secure systems in a cost effective manner. It includes the "Core" Tokeneer ID System Software, test cases derived from the system test specification, "Support" Tokeneer ID System Software and test tokens and biometric data, project documents. Since the delivery, the Tokeneer project has become a milestone point and a testing range for different verification tools before applying them in industrial projects. Despite the fact that after delivery 4 bugs<sup>1</sup> were found, the system is still deemed to be very secure.

Tokeneer is a secure enclave consisting of a set of system components, some housed inside the enclave and some outside, as depicted in Figure 1. The ID Station (TIS) is part of the larger Tokeneer system. It has four connected peripherals, namely, a fingerprint reader, a smartcard reader (users use Tokens -smartcards- as identification), a door and visual display. The objective of the enclave is to ensure that anyone who enters the enclave has a proper access, and no one else can access to the enclave.

<sup>1</sup>According to [7]

In order to ensure the entrance of users to the enclave, TIS implements a series of protocols and checks (the use of smart cards and biometrics) to grant or deny the entrance to it. This paper discusses one of these protocols: the Enrollment to the ID Station. The protocol starts in a state where the user is not enrolled. Users can request enrollment and then insert a FLOPPY (it retains an internal view of the last data written) for the system to proceed. The system reads the information in the floppy and either fails the enrollment process, in which case takes the process to the initial state, or correctly validates the data in the floppy.

### B. Eiffel

Eiffel is a real complex object oriented programming language that natively supports Design-by-Contract methodology. Users can specify the behavior of Eiffel classes by equipping them with contracts: pre- and post-conditions and class invariants, that are represented as assertions.

```

class
ACCOUNT
create make

feature -- Initialization
make -- Initialize empty account.
do
    balance := 0
ensure
    balance_set: balance = 0
end

feature -- Access
balance : INTEGER -- Balance of account.

feature -- Element change
deposit (amount : INTEGER)
-- Deposit 'amount' on account.
require
    amount_not_negative : amount >= 0
do
    balance := balance + amount
ensure
    balance_increased : balance =
        old balance + amount
end

withdraw (amount : INTEGER)
-- Withdraw 'amount' from account.
require
    enough_balance : amount <= balance
do
    balance := balance - amount
ensure
    balance_decreased : balance = old balance - amount
end

invariant
    non_negative_balance : balance >= 0
end

```

Fig. 2. ACCOUNT Eiffel class

Figure 2 depicts a reduced implementation of a Bank Account. In Eiffel, creation procedures are listed under the keyword **create**, for class **ACCOUNT**, routine **make** is used as a creation procedure. The class defines a class attribute **balance** to represent the current balance of the account. It also defines two routines (methods), **deposit** and **withdraw**. **deposit** implements a deposit of amount **amount** of money to the account and **withdraw** implements withdrawing money. Eiffel encourages software developers to express formal properties of classes by writing assertions. Routine pre-conditions express the requirements that clients must satisfy whenever they call a routine. They are introduced in Eiffel by the keyword **require**. Routine **deposit** imposes a pre-condition on the call, the client must pass as an argument a non-negative number (i.e. **amount\_not\_negative: amount >= 0**) for the routine to work correctly: a negative value might invalidate the invariant of the class. Routine post-conditions, introduced in Eiffel by the keyword **ensure**, express conditions that the routine (the supplier) guarantees on method exit, assuming the pre-condition. Routine **deposit** guarantees that the balance of the account will be the previous value of the balance (expressed in Eiffel by the keyword **old**: the value on entrance of the routine) plus the amount being deposited. Routine **withdraw** imposes the constraint to the caller that the argument must be less than or equal to the current balance of the account to avoid having negative value in the balance. The routine ensures that, after execution, the new value of **balance** will be the value on routine entry minus the amount withdrawn.

A class invariant must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class (public routines). The invariant appears in a clause introduced by the keyword **invariant**. Class **ACCOUNT**'s invariant imposes the restriction that class attribute **balance** can never be negative (i.e. **non\_negative\_balance: balance >= 0**).

### C. AutoProof

AutoProof [5] is a static verifier of contracts for Eiffel programs. It follows the auto-active paradigm where verification is done completely automated, similar to model checking [3], but users are expected to feed the classes providing additional information in the form of annotations to help the proof. AutoProof identifies software issues without the need of executing the code, therefore opening a new frontier for "static debugging", software verification and reliability, and in general for software quality.

AutoProof verifies the functional correctness of Eiffel classes. It translates Eiffel code to Boogie programs [10] and calls the Boogie tool to generate verification conditions: logic formulas whose validity entails correctness of the input programs. Finally, retrieves the answer back to Eiffel. AutoProof verifies that routines satisfy pre- and post-conditions, maintenance of class invariants, loops and recursive calls termination, integer overflow and non **Void** (*null* in other programming languages) references calls. The tool also supports most of

```
class
  ID_STATION
  ... Some lines were omitted. ...
create
  make
feature -- Initialization
  make
  note
    status : creator
  do
    ... Some lines were omitted. ...
  end
end
```

Fig. 3. Initialisation of **ID\_STATION** Eiffel class.

the Eiffel language constructs: in-lined assertions such as **check** (*assert* in other programming languages), types, multi-inheritance, polymorphism.

### III. VERIFICATION OF TOKENEER USING AUTOPROOF

The Tokeneer project was implemented in Eiffel following the specifications file 41\_2.pdf (see [7]) of the Tokeneer System and equipping classes with contracts. This research work encompasses only the enrolment process of the whole Tokeneer System therefore it implements only the entities involved in this process.

One of the main parts of TIS is the **ID\_STATION** (see Figure 8) - it describes how all components of the system are related to each other: one of the components is implemented in class **INTERNAL\_S** (not shown here) whose responsibility is to keep knowledge of the status of user entry and the enclave and to hold a timeout when relevant; another component is implemented on class **FLOPPY** (not shown here) that retains an internal view of the last data written to the floppy as well as the current data on the floppy. **ID\_STATION** displays the configuration data on the screen which is implemented in **SCREEN\_DISPLAY**. There are a number of messages that may appear on the TIS screen. The Real World types (described in [7] Specification document, section 2.7.1) of the system such as messages that appear on the display and screen, were implemented all together in class **CONST** which implements the constants used in the TIS. And finally, a number of interactions between all these entities within the enclave are implemented in **ENCLAVE\_OPERS**.

AutoProof does not make any assumptions out of box therefore users are expected to feed the Eiffel classes for a succeed verification. This is expressed by means of Eiffel's **note** clause. **note** clause enables users to attach addition information to the class that is ignored by the Eiffel's compiler. AutoProof uses this information to succeed in the verification. For instance, Autoproof's annotation **status** defines which procedure is used to initialize newly created objects: Figure 3 depicts procedure **make** with annotation **note** (e.g. **note status: creator**) to help Autoproof to discharge the corresponding proof obligations related to creation procedures: the procedure will be called

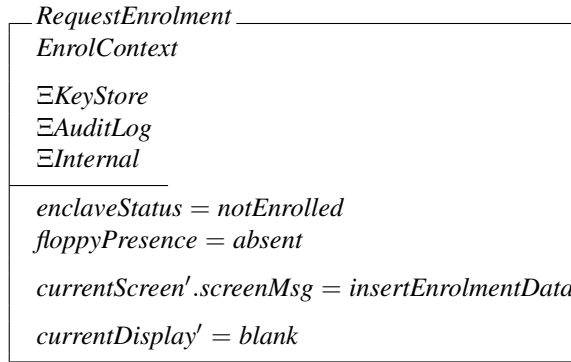


Fig. 4. Z schema of RequestEnrolment

```

make
... Some lines were omitted. ...
ensure
  enclave_status = cons_floppy.not_enrolled
  floppy_presence = cons_internal.absent
  token_removal_timeout = 0
end

```

Fig. 5. ensure clause in feature make

only when an object of this class is being created, AutoProof needs to verify a creation routine only once.

**note** clause is also used to define models queries to express the abstract state space of a classes. Model queries are part of model-based contracts to help users to write abstract and concise specifications [11], they are used to specify the behavior of the class. In Eiffel, this is specified by adding a **note** clause at the beginning of the class followed with a keyword **model:** and listing one or more attributes of the class. Model queries are also used to describe frame conditions: which allocations are allowed to be modified by procedures. In Eiffel, frame conditions are listed using the **modify** clause, which lists the model queries that the feature is allowed to modify, as shown in Figure 7 (i.e. `modify_model("current_display", Current)`).

According to RequestEnrolment (a Z-schema that is a part of the formal specification of the project Tokeneer), which is presented in Figure 4, requesting enrolment involves EnrolContext, KeyStore, AuditLog, Internal. Schemas in Z consist of an upper part, in which some variables are declared, and a lower part, which describes the relationship between values and variables. The notation  $\Xi$  indicates an operation in which the state does not change, and the apostrophe indicates the state of the variable after the change [12]. RequestEnrolment specifies that the ID station will request enrolment by displaying a request string on the screen and keeping the display blank. This is only possible while there is no Floppy present. Therefore, initially **floppyPresence** = **absent** and **enclaveStatus** set to **notEnrolled**. An **ensure** clause was used in the creation procedure to guarantee this after the initialization of **ID\_STATION** object:

```

invariant
  constants.display_message.has(current_display)
  constants /= Void

```

Fig. 6. Invariants of ID\_STATION Eiffel class.

```

feature --Element Change
  set_current_display(v: STRING)
  require
    constants.display_message.has(v)
    modify_model("current_display", Current)
  do
    current_display := v
  ensure
    current_display = v
  end

```

Fig. 7. Feature equipped with **modify** clause

Figure 6 depicts the class invariant for class **ID\_STATION**. It states that a message displayed on the display outside the enclave is one of the available from the list of messages (i.e. `constants.display_message.has(current_display)`) and that class attribute `constants` is attached to an object (i.e. `constants /= Void`).

Figure 7 shows the implementation of procedure **set\_current\_display**. Its first pre-condition was added to satisfy the invariant ensuring that argument *v* belongs to the allowed displayed messages. The second pre-condition restricts the procedure to change values only to model query **current\_display**.

Figure 8 shows the final version of class **ID\_STATION**: with the respective annotations for Autoproof to successfully verify the class. In class **ID\_STATION**, class attributes `current_screen` and `current_display` implements the physical screen and display, respectively, of the enclave.

#### IV. EMPIRICAL RESULTS

The usability of a verification tool cannot be considered in isolation and, in particular, cannot be hived off by the effectiveness of the tool itself. First, as a general observation, the cost of using an instrument can only be justified by its return, which can ultimately be linked to financial consideration by top management. Second, and this aspect is less general and more peculiar to the auto-active verification approach, a tool like AutoProof is as much effective and usable as is its ability to discharge verification conditions completely automatically, without feeding the code of annotation overhead or requiring particular tweaking. Finally, the necessity for users to add further annotations and dedicate extra effort (and considerable time) is, by itself, an obstacle to adoption and (technically) an usability issue. Verification tools should require minimal annotational effort and give valuable feedback when verification fails.

The case study analyzed in this paper presented good results in term of automatic discharge of verification conditions,

```

class
  ID_STATION
  ... Some lines were omitted. ...

create
  make

feature -- Initialization
  make
  note
    status: creator
  do
    create constants
    current_display := constants.blank
    create current_screen.make

    create cons_floppy
    enclave_status := cons_floppy.not_enrolled
    token_removal_timeout := 0

    create cons_internal
    floppy_presence := cons_internal.absent
  ensure
    enclave_status = cons_floppy.not_enrolled
    floppy_presence = cons_internal.absent
    token_removal_timeout = 0
  end

feature -- Element Change
  set_current_display(v: STRING)
  require
    constants.display_message.has(v)
    modify_model("current_display", Current)
  do
    current_display := v
  ensure
    current_display = v
  end

feature -- Access
  constants : CONST
  current_screen : SCREEN_DISPLAY
  current_display : STRING
invariant
  constants.display_message.has(current_display)
  constants /= Void
end

```

Fig. 8. Verified **ID\_STATION** Eiffel class.

though not comparable to others seen in literature [13]. In total there were 38 generated proof. Of these, 22 (58%) were discharged automatically, 8 (21%) could not be satisfied, and the rest (21%) failed due to internal errors, which in our case were basically caused by the attempt to create objects in the contract, and that is not allowed by the tool. As observed before, the success of verification is unsurprisingly linked to the complexity of programs [13]. Previous literature mostly dealt with students users and university projects. The use of Tokeener as a benchmark demands for detailed comparisons with different verification efforts (for example, [14]).

## V. RELATED WORK

Formal/mathematical notations have existed for a long time and have been used to specify and verify systems. Examples are process algebras [15], specification languages like Z [16], B [17] and Event-B [18]. The Vienna Development Method (VDM) is one of the earliest attempts to establish a formal method for the development of computer systems [19]. A survey of these (and others) formalisms can be found in [20] while a discussion on the methodological issues of a number of formal methods is presented in [21].

All these approaches (and others described in the literature) still leave an open issue, i.e., they are built around strict formal notations which affect the development process from the very beginning. These approaches demonstrate a low level of flexibility. To overcome this problem, a seamless methodological connection built on top of a portfolio of diverse notations and methods is presented in [22]. Another approach is presented in [14], [23] using [24], where users start the development of system from a strict formal notation (i.e. Event-B), to then automatically translate it to Java code with JML [25] specifications embedded (following Design-by-Contract methodology). Even though this approach enables users with less mathematical expertise to work on formal development, it does not give a seamlessly methodology for the development as presented in this paper.

On the other side, Design-by-contract [6] when combined with AutoProof technology offers the pros of both rigorous methodologies and supporting tools able to semi-automate the process. Before this to be available for the average developer it is however necessary to improve the users' experience. A comparison between different approaches (for example Event-b/Rodin and Design-by-contract/AutoProof) is beyond the scope of this paper and it is left as future work.

## VI. CONCLUSION

AutoProof allows for "static debugging", i.e. debugging becomes possible without the need of executing the program. The most effective way to release correct software is a combination of static debugging and traditional run-time debugging. Being all human activities (therefore including programming and testing itself) error-prone, there is no magic or free lunches out there. Abandoning testing and adopting a proof-oriented approach does not make miracles, debugging remains a trial-and-error long and laborious process. AutoProof does not change the rules of the game: developers will have to try, observe the results and make changes as a consequence. A proof-oriented approach does not make the process smoother and necessarily simpler. However, it makes it more accurate and robust, therefore effective. Adjustment can be now focused on the the implementation side (possibly sinergically with run-time debugging), on the specification side (the contracts used to annotate the code as integral part of the code itself), or in the proof itself (fine-tuning may be necessary for AutoProof and its behind-the-curtains machinery to be able to prove correctly).

All this comes with a cost: the willingness and ability of the user to use extra tools and being able to master them, and possibly invest extra time in the process. On the other side, it is necessary for the tools to be simple to master and to provide intelligible feedback.

The Tokeneer project case study showed the efficacy of AutoProof in verifying a real piece of software, the complexity of which can be compared not only with most of the commercial Off-the-Shelf software, but also with safety and financial-critical applications, both in terms of computational logic and architectural organization. AutoProof is capable to verify industrial software and may well be adopted in commercial companies and its use injected into the development process. However, some obstacles have been identified that could prevent its broader adoption.

As result of an academic effort, documentation is not at par with commercial software, in particular for what concerns the size of the library of correctly verified examples: tutorials on the official website are quite useful, but not enough. On top of this, the tool itself has limitations. First, existing implementations need to be modified in order to be verified. This would represent an unsurmountable obstacle in most institutions since the overall cost of code adaptation may overrun the saves occurring to the testing phase. This consideration may be different, however, for safety-critical systems. Second, the Eiffel IDE - necessary for functioning - calls for increased stability and robustness.

#### ACKNOWLEDGMENTS

We would like to thank Innopolis University for logistic and financial support, and the laboratories of Software Engineering (SE) and Service Science and Engineering (SSE) for the intellectual engagement and vivid discussions.

#### REFERENCES

- [1] B. Meyer, *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer Publishing Company, Incorporated, 1 ed., 2009.
- [2] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix-points," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977.
- [3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [4] D. W. Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*. sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.
- [5] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, "Autoproof: Auto-active functional verification of object-oriented programs," in *21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 566–580, 2015.
- [6] B. Meyer, *Object-oriented software construction*, ch. 11: Design by Contract: building reliable software. Prentice Hall PTR, 1997.
- [7] AdaCore, "Tokeneer." <http://www.adacore.com/sparkpro/tokeneer/download>, accessed in April 2016.
- [8] J.-R. Abrial, S. Schuman, and B. Meyer, "Specification Language," in *On the Construction of Programs*, R. M. McKeag and A. M. Macnaghten, editors, pp. 343–410, Cambridge University Press, 1980.
- [9] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
- [10] K. R. M. Leino, "This is boogie 2," tech. rep., June 2008.
- [11] N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying reusable components," in *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)* (G. T. Leavens, P. O'Hearn, and S. Rajamani, eds.), vol. 6217 of *Lecture Notes in Computer Science*, pp. 127–141, Springer, August 2010.
- [12] J. Spivey, "An introduction to Z and formal specifications," *Software Engineering Journal*, 1989.
- [13] C. A. Furia, C. M. Poskitt, and J. Tschannen, "The autoproof verifier: Usability by non-experts and on standard code," in *Proc. Formal Integrated Development Environment (F-IDE 2015)*, vol. 187, pp. 42–55, Electronic Proceedings in Theoretical Computer Science (EPTCS), 2015.
- [14] V. Rivera, S. Bhattacharya, and N. Cataño, "Undertaking the tokeneer challenge in Event-B," *To appear in 4th FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2016.
- [15] J. C. M. Baeten, "A brief history of process algebra," *Theor. Comput. Sci.*, vol. 335, no. 2-3, pp. 131–146, 2005.
- [16] J. Abrial, S. A. Schuman, and B. Meyer, "Specification language," in *On the Construction of Programs*, pp. 343–410, 1980.
- [17] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
- [18] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 1st ed., 2010.
- [19] C. B. Jones, *Software Development: A Rigorous Approach*. Englewood Cliffs, N.J., USA: Prentice Hall International, 1980.
- [20] M. Mazzara and A. Bhattacharyya, "On modelling and analysis of dynamic reconfiguration of dependable real-time systems," in *Proceedings of the 2010 Third International Conference on Dependability, DEPEND '10*, (Washington, DC, USA), pp. 173–181, IEEE Computer Society, 2010.
- [21] M. Mazzara, "Deriving specifications of dependable systems: toward a method," in *Proceedings of the 12th European Workshop on Dependable Computing, EWDC*, 2009.
- [22] R. Gmehlich, K. Grau, A. Iliasov, M. Jackson, F. Loesch, and M. Mazzara, "Towards a formalism-based toolkit for automotive applications," *1st FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2013.
- [23] V. Rivera, N. Cataño, T. Wahls, and C. Rueda, "Code generation for Event-B," *To appear in International Journal on STTT*, 2016.
- [24] V. Rivera and N. Cataño, "Translating Event-B to JML-Specified Java programs," in *29th ACM SAC*, (Gyeongju, South Korea), March 24–28 2014.
- [25] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: A behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–38, May 2006.

# Certified Grammar Transformation to Chomsky Normal Form in F\*

Marina Polubelova  
Saint Petersburg State University  
St. Petersburg, Russia  
Email: polubelovam@gmail.com

Sergey Bozhko  
Saint Petersburg State University  
St. Petersburg, Russia  
Email: gkerfimf@gmail.com

Semyon Grigorev  
Saint Petersburg State University  
St. Petersburg, Russia  
Email: Semen.Grigorev@jetbrains.com

**Abstract**—Certification programming allows to prove that the program meets its specification. The check of correctness of a program is performed at compile time, which guarantees that the program always works as specified. Hence, there is no need to test certified programs to ensure they work correctly. Among the toolchains designed for certified programming, F\* is the only language that support semi-automated proving and general-purpose programming. We work on the application of this technique to a grammarware research and development project YaccConstructor. We present a verified implementation of transformation of Context-free grammar to Chomsky normal form, that is making progress toward the certification of the entire project. We also discuss advantages and disadvantages of such approach and formulate topics for further research.

**Keywords**—certified programming; F\*; program verification; context-free grammar; Chomsky normal form; grammar transformation; dependent type; refinement type

## I. INTRODUCTION

Certified programming is designed for proving that a program meets its specification. For this technique, proof assistants or interactive theorem prover are used [1], what allows to check correctness of the program at compile time and guarantees that the program always works according to its specification. Classical fields of application of certified programming are the formalization of mathematics, security of cryptographic protocols and the certification of properties of programming languages.

There are two approaches to certified programming [2]. In the classical approach the program, its specification, and the proof that the program meets its specification are written separately, as different modules. Such technique costs too much to be applied in software development. More effective approach is to combine program, its specification, and the proof in one module by means of dependent types [3], [4]. The most well-known toolchains for program verification are Coq [5], Agda [6], F\* [7] and Idris [8]. Among them, F\* is the only language which supports semi-automated proving and general-purpose programming.

As a proof assistant, F\* allows to formulate and prove properties of programs by using lemmas and enriching types. F\* not only infers types of functions, but also the properties of its computations such as purity, statefulness, divergence. For example, consider the following function:

```
val f : (int -> Tot int) -> int -> Tot int
let f g x = g x
```

The keyword `val` indicates that we declare a function `f` and its type signature. The function `f` takes a function `g` and an integer value, as arguments. The effect of computation `Tot t` is used for total expression which always evaluates to a `t`-typed result without entering an infinitive loop, throwing exception or other side effects. Hence, one can prove for some programs not only their properties and restrictions on the types, but also guarantee their termination and that a result has assigned type.

We apply certified programming using F\* to a grammarware research and development project YaccConstructor (YC) [9], [10]. YC is a tool for parser construction, grammar processing, parser generators and other grammarware for .NET platform. The verification of its programs covers the topic of parser correctness: how to obtain formal evidence that a parser is correct with respect to its specification [11].

In this article, we consider only one algorithm implemented in YC, namely the transformation of context free grammar to Chomsky normal form, that is a small step towards the certification of entire project. The algorithm of grammar normalization consists of four transformations. We prove totality of each of them and establish an order of their application to the input grammar. In addition, we describe the peculiarities of evaluation F\* as a proof assistant and formulate topics for further research.

## II. OVERVIEW OF F\*

We use a functional programming language F\* [7] for program verification. It is the only language that support semi-automated proving and general-purpose programming [12]. The main goal of this tool is to span the capabilities of interactive proof assistants like Coq [5] and Agda [6], general-purpose programming languages like OCaml and Haskell, and SMT-backed semi-automated program verification tools like Dafny [13] and WhyML [14].

Type system of F\* includes polymorphism, dependent types, monadic effects, refinement types, and a weakest precondition calculus [15], [16]. These features allow expressing precise and compact specification for programs [7].

- *Dependent function type* has the following form:

$$x_1 : t_1 \rightarrow \dots \rightarrow x_n : t_n[x_1 \dots x_{n-1}] \rightarrow E \ t[x_1 \dots x_n].$$

Each of a function's formal parameters are named  $x_i$  and each of these names in the scope to the right of the first arrow that follows it. The notation  $t[x_1 \dots x_m]$  indicates that the variables  $x_1 \dots x_m$  may appear free in  $t$ .

- *Refinement type* has a form  $x : t\{\text{phi}(x)\}$ . It is a sub-type of  $t$  restricted to those expression of type  $t$  that satisfy the predicate  $\text{phi}(e)$ .
- In addition to inferring a type, F\* also infers side effects of an expression such as exceptions, state. We consider the main monadic effects.
  - $\text{Tot } t$  — the effect of a computation that guarantees evaluation to a  $t$ -typed result, without entering an infinite loop, throwing an exception, reading or writing the program's state.
  - $\text{ML } t$  — the effect of a computation that may have arbitrary effects, but if some result is computed, then it is always of type  $t$ .
  - $\text{Dv } t$  — the effect of a computation that may diverge.
  - $\text{ST } t$  — the effect of a computation that may diverge, read, write or allocate on a heap.
  - $\text{Exn } t$  — the effect of a computation that may diverge or raise an exception.

The effects  $\{\text{Tot}, \text{Dv}, \text{ST}, \text{Exn}, \text{ML}\}$  are arranged in a lattice, with  $\text{Tot}$  at the bottom,  $\text{ML}$  at the top, and with  $\text{ST}$  unrelated to  $\text{Exn}$ .

There are two main approaches to prove properties: either by enriching the type of a function (intrinsic style) or by writing a separate lemma about it (extrinsic style). You can see an example of the first approach, in which the keyword `val` indicates declaration of a value and its type signature, below.

```
val append: 11:list 'a -> 12:list 'a
-> Tot (1:list 'a{length 1=length 11+length 12})

let rec append 11 12 =
match 11 with
| [] -> 12
| hd :: tl -> hd :: append tl 12
```

The following example demonstrates extrinsic style, in which the formula after keyword `requires` is the precondition of the function/lemma, while the one after keyword `ensures` is its post-condition:

```
val append_len: 11:list 'a -> 12:list 'a
-> Lemma (requires True)
  (ensures (length (append 11 12) =
    length 11 + length 12)))

let rec append_len 11 12 =
match 11 with
| [] -> ()
| hd::tl -> append_len tl 12
```

There is no rule which style of proving to use, but in some cases it is impossible to prove a property of a function directly in its types and one has to use a lemma.

When defining lemmas or expressions that are total, F\*

automatically proves their termination. The termination check is based on a well-founded relation. For natural numbers, F\* uses classical decreasing metric, for inductive types — the sub-term ordering, for recursive function, it requires the tuple of parameters to be in decreasing lexicographic ordering. The last case can be overridden with using clause `decreases %[x1; ...; xn]`, which explicitly chooses a lexicographic ordering on arguments.

So, we can use F\* to write effectful programs, to specify them using dependent and refinement types, to verify them using an SMT solver or providing interactive proofs. Programs written in F\* can be translated to OCaml or F# for further execution.

### III. VERIFICATION OF TRANSFORMATION OF CFG TO CNF

In this section we briefly describe a part of theory of formal languages used, the proof of totality of one of transformations of grammar to Chomsky normal form in F\*, and formulate some advantages and disadvantages of this approach.

#### A. Context-free grammar and Chomsky normal form

In this section we give basic definitions and formulate a theorem that helps us to verify implemented algorithm of transformation of context-free grammar to Chomsky normal form.

In formal language theory, a **context-free grammar** (CFG) is a formal grammar in which every production rule is of the form  $A \rightarrow \alpha$ , where  $A$  is single nonterminal symbol and  $\alpha$  is a string of terminals and/or nonterminals ( $\alpha$  can be empty).

Context-free grammar  $F$  is said to be in **Chomsky normal form** (CNF) if all of its production rules are of the form:

- $A \rightarrow BC$
- $A \rightarrow a$
- $S \rightarrow \epsilon$ ,

where  $A, B$  and  $C$  are nonterminal symbols,  $a$  is a terminal symbol,  $S$  is a start nonterminal, and  $\epsilon$  denotes the empty string. Also, neither  $B$  nor  $C$  may be the start symbol, and the third production rule can only appear if  $\epsilon$  is in  $L(G)$ , namely, the language produced by the context-free grammar  $G$ .

Context-free grammars given in Chomsky normal form are very convenient to use. It is often assumed that either CFGs are given in CNF from the beginning or there is an intermediate step of normalization. Having a certified implementation of normalization for CFGs enables us to stop thinking in terms of CFG and consider grammar in CNF without losing guarantees of correctness.

**CFG normalization theorem:** There is an algorithm which converts any CFG into an equivalent one in Chomsky normal form.

The full normalization transformation for a CFG is the composition of the following constituent transformations.

- Replacing all rules  $A \rightarrow X_1 X_2 \dots X_k$ , where  $k \geq 3$  with rules  $A \rightarrow X_1 A_1$ ,  $A_1 \rightarrow X_2 A_2$ ,  $\dots$ ,  $A_{k-2} \rightarrow X_{k-1} X_k$ , where  $A_i$  are “fresh” nonterminals.
- Elimination of all  $\epsilon$ -rules.



- Elimination all chain rules.
- For each terminal  $a$ , adding a new rule  $A \rightarrow a$ , where  $A$  is a “fresh” nonterminal and replacing  $a$  in the right-hand sides of all rules with length at least two with  $A$ .

### B. Verification with $F^*$

Our purpose is a verification of programs in YaccConstructor (YC) using  $F^*$ . YC is an open source modular tool for research in lexical and syntax analysis and its main development language is F# [17]. In this paper we consider only a verification of normalization grammar algorithm [18], which in YC looks like:

```
let toCNF (ruleList: Rule.t<_,>list) =
    ruleList
    |> splitLongRule
    |> deleteEpsRule
    |> deleteChainRule
    |> renameTerm
```

The function `toCNF` is a composition of the four transformations mentioned. Notice that the order of rule execution is important. The first rule must be executed before the second, otherwise normalization time may increase to  $O(2^n)$ . The third rule follows the second, because elimination of  $\epsilon$ -rules may produce new chain rules. Also, the fourth rule must be executed after the second and the third as they can generate useless symbols.

$F^*$  as a proof assistant allows to formulate and prove properties of function of interest using lemmas or enriching types. For example, in F# function `(f (x:int) = 2*x)` is inferred to have type `(int -> int)`, while in  $F^*$  we infer `(int -> Tot int)`. This indicates that `(f (x:int) = 2*x)` is a pure total function which always evaluates to `int`. A lemma is a ghost total function that always returns the single unit value `()`. When we specify a total function, we have to prove totality of every nested function, because  $F^*$  supports only high-level annotations. In other words, we cannot add annotation for nested function. Therefore, to prove totality of a function containing nested function, we need to lift all nested functions up and explicitly prove totality of these functions.

We describe each function of interest in an individual module to avoid namespace collision. We use module architecture similar to YC architecture. We have module `IL`, that contains type constructors for describing rules, productions and etc. Also, there is a module `Namer`, which contains a function to generate new names. Finally, there are individual modules for each transformation, and a separate, main, module which contains the definition of `toCNF` transformation.

$F^*$  does not provide any primitive support for object-oriented features. That is why records became the main structure, used in the implementation. Hence, as records do not have constructors, for each of them we created functions that generate elements of the type directly; some of these functions described in module `TransformAux`. In other words, rather than create class `Person` with constructors and methods like this:

```
let person = new Person("Nick", 27)
```

We have to do it the following way:

```
let new_Person name age = {name=name; age=age}
```

We rewrote all the transformations in  $F^*$ , but in this paper we consider only one of them, namely `SplitLongRules` which eliminates long rules. Firstly, we describe all the helpers we need, prove their totality and other necessary properties, and then explain why this transformation is correct.

In the first transformation it is necessary to create new nonterminals, so we need a function to supply them. The function `Namer.newSource` defined below is used.

```
val newSource:
    n:int->oldSource:Source->Tot Source
let newSource n old =
    ({old with text=old.text^(string_of_int n)})
```

Integer  $n$  is equal to the size of the list of rules which we have at the moment of function `Namer.newSource` call. Obviously, function `Namer.newSource` is injective. In other words, if we have unique rule names, they remain unique after application `splitLongRule`. Some necessary helpers are grouped in `TransformAux` module: for example, functions `createRule` and `createDefaultElem`, which take some arguments and return `Rule` and `Elem` respectively. `Elem` is the right part of the rule if the latter is a sequence. Also, we define follow one simple function which returns the length of the right part of the rule.

```
val lengthBodyRule:Rule 'a 'b->Tot int
let lengthBodyRule rule =
    List.length (match rule.body with
        | PSeq(e, a, l) -> e
        | _ -> [ ])
```

The most interesting function is `cutRule`. It takes a rule and a list-accumulator as an input. If the length of the right part of the rule is less or equal to 2, we do not need to do anything and `cutRule` only renames a nonterminal to avoid name collision. Otherwise we have to create new nonterminal  $B_{k-2}$ , cut off last two elements  $X_{k-1}X_k$ , pack them into a new rule  $B_{k-2} \rightarrow X_{k-1}X_k$ , and then add the nonterminal to the end of the long rule. Then the new rule is added to the accumulator and the function `cutRule` is recursively called on the new rule and accumulator. This way, we reduce our rule by one. Function signature is the following.

```
val cutRule: rule:(Rule 'a 'b)
-> resRuleList:(list (Rule 'a 'b))
-> Tot (list (Rule 'a 'b))
    (decreases %[lengthBodyRule rule;
        List.length resRuleList])
```

There are some peculiarities in our implementation which are worth mentioning. One of them is the representation of the right part of the rules by lists. In the algorithm we need to cut off two last elements of a rule, so we carry out the following steps.

```
let revEls = List.rev elements
let cutOffEls = [List.Tot.hd revEls;
```

```
List.Tot.hd (List.Tot.tl revEls)]
```

Standard functions `List.hd` and `List.tl` are not defined for an empty list, but we need to use total functions. In  $F^*$  there is a module `List.Tot`, in which these functions are described. We only give their signature here.

```
val hd: l:list 'a{is_Cons l}->Tot 'a
val tl: l:list 'a{is_Cons l}->Tot (list 'a)
```

Where `is_Cons` takes a list as an input and returns `false` if it is empty, otherwise it returns `true`.

It means that if we try to apply function `List.Tot.hd` to a list, nonemptiness of which is not clear from the context,  $F^*$  reports a type mismatch. A pleasant peculiarity of  $F^*$  is that in some rare cases it can derive necessary properties. We divide only the rules which have more than two symbols in the right side. In this case  $F^*$  is able to automatically derive required type, so we can choose two elements. It can be explained in the following example.

```
// lst has type list int and can be empty
assume val lst: list int
// f takes only nonempty lists
assume val f:
lst:(list int){is_Cons lst}->Tot int
assume val g: lst:(list int)->Tot int

//Ok
let test1 (lst:list int) =
  if List.length lst >= 1
  then f lst
  else g lst

//Fail: subtyping check failed
let test2 (lst:list int) =
  if List.length lst >= 0
  then f lst
  else g lst
```

At the same time, we have to prove and explicitly add even simple lemmas for functions. For example, if list `lst` has type `(list 'a){is_Cons lst}`, then `(List.rev lst)` only has type `(list 'a)`. This can be easily fixed with the instruction `SMTPat`. In addition, we should formulate the following lemma which proof can be derived automatically by  $F^*$ . The following code makes `List.rev` preserve information about the length:

```
val rev_length:
l:(list 'a)
-> Lemma (requires true)
(ensures
(List.length (List.rev l)=List.length l))
[SMTPat (List.rev l)]
```

We proved totality of all the nested functions. Now we want to prove termination of the general one. In our case, it is sufficient that the length of the rule strictly decreases on each recursive call and we are not interested in the length of the accumulator. To prove this we must explicitly specify that after applying `List.Tot.tl` to a list, its length reduces by 1. So, we must use the same method as we used before.

```
val tail_length :
l:(list 'a){is_Cons l}
-> Lemma (requires True)
(ensures
(List.length (List.Tot.tl l)=(List.length l)-1))
[SMTPat (List.Tot.tl l)]
```

With this sufficient information  $F^*$  has to conclude that `cutRule` is total.

Function `splitLongRule` takes a list of rules and applies `(fun rule -> cutRule rule [ ])` to each rule, then concatenates all the results and returns the combined list.

```
val splitLongRule: list (Rule 'a 'b)
-> Tot (list (Rule 'a 'b))
let splitLongRule ruleList =
  List.Tot.collect
    (fun rulecutRule rule [ ] ruleList
```

Totality is proved automatically by  $F^*$ .

Previously we proved totality of our transformation, but we had not mentioned properties of the rules we get after applying `splitLongRule`. We add restriction on the type of function, which guarantees the necessary property of the result, instead of proving the lemma about these properties. The function signature now look like this.

```
val cutRule: rule:(Rule 'a 'b)
-> acc:(list (Rule 'a 'b))
{List.Tot.for_all
  (fun x->lengthBodyRule x<=2) acc}
-> Tot (res:(list (Rule 'a 'b))
{List.Tot.for_all
  (fun x->lengthBodyRule x<=2) res})
(decreases %[lengthBodyRule rule;
  List.length resRuleList])

val splitLongRule:list (Rule 'a 'b)
-> Tot (res:list (Rule 'a 'b)
{List.Tot.for_all
  (fun x->lengthBodyRule x<=2) res})
```

Now we have almost everything we need to prove such properties. We have to add some more information so that  $F^*$  could check arguments type when recursively called. At the moment of cutting the rule off, we should fix the length in the type of the cut part. For this purpose we have to define a function to take our list and return part with that type. Further, we have to prove lemma that states that concatenation of two lists with short rules is a list with short rules. After that  $F^*$  accepts type correctness.

### C. Advantages and disadvantages of $F^*$

In  $F^\#$ , even if we have some correct functions, we may get incorrect result by applying them in a wrong order. An advantage of  $F^*$  is that it can prevent such situations, if we specify the properties we demand from an input data in a type of a function. For instance, we can apply `deleteChainRule` only after deleting all the epsilon rules. The signature of `deleteChainRule` is described in the following way.

```
val deleteChainRule:
ruleList:(list (Rule 'a 'b))
```

```
{is_non_eps_rules ruleList}
-> Tot (list (Rule 'a' 'b'))
```

Unfortunately, there are some significant disadvantages of F\*. It would be better if F\* was more automated, because one can usually realize totality of a function far earlier than F\*. Sometimes it is hard to understand why proofs do not pass correctness tests. The reason is that F\* does not give us any information about errors. There are subgoals in Coq, which allow user to create a proof interactively, from a hypothesis to prove, and with the help of tactics he divides it to subgoals, which let us understand proof process. Such mechanism would be useful in F\*. There are a special construct in many functional languages which checks whether some property holds for a value. Such construct is called `guard` in Haskell and `when` in OCaml and F# and is often used in pattern matching to simplify code. Unfortunately, it is not supported in F\* and one could only hope that it will be supported in the latter language versions.

#### IV. CONCLUSION AND FUTURE WORK

We presented a verification of some transformations of context-free grammar to the Chomsky normal form. The purpose of that is a proof of totality of each function used. This property guarantees that computations always terminate and do not enter in infinite loop, throw exception and don not have other side effects. Although a correctness proof of the rest of transformations is still in progress, we have significant results. We can specify an input and an output of functions, using refinement and dependent types, that allow us to establish an order of application of the four transformations. So, when we prove totality of each function, we prove that an entire conversion of grammar is total which is very useful in practice.

For verification of considered algorithm, we use a programming language F\* which only allows to verify the input code, but not to execute it. To execute F\* code one needs to extract it to OCaml or F# and then compile it using the OCaml or F# compiler respectively. At the moment, the mechanism of extraction code from F\* to F# omits casts, erases dependent types, higher rank polymorphism and ghost computation [12]. These features are very important and lack of them breaks the consistency and correctness of programs within the target language. As far as we know, F\* is a language which is still under development, and implementation of the extraction mechanism, which cope with the above shortcoming, is actual topic for our further research.

#### REFERENCES

- [1] H. Geuvers, "Proof assistants: History, ideas and future," *Sadhana*, vol. 34, no. 1, pp. 3–25, 2009.
- [2] A. Chlipala, "Certified programming with dependent types," 2016.
- [3] T. Sheard, A. Stump, and S. Weirich, "Language-based verification will change the world," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 343–348.
- [4] E. Tanter and N. Tabareau, "Gradual certified programming in Coq," in *Proceedings of the 11th Symposium on Dynamic Languages*. ACM, 2015, pp. 26–40.
- [5] The Coq proof assistant. [Online]. Available: <https://coq.inria.fr/>
- [6] The Agda homepage. [Online]. Available: <http://wiki.portal.chalmers.se/agda/pmwiki.php>

- [7] The F\* homepage. [Online]. Available: <https://www.fstar-lang.org/>
- [8] The Idris homepage. [Online]. Available: <http://www.idris-lang.org/>
- [9] The YaccConstructor homepage. [Online]. Available: <https://github.com/YaccConstructor/>
- [10] I. Kirilenko, S. Grigorev, and D. Avdiukhin, "Syntax analyzers development in automated reengineering of informational system," *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, no. 174, pp. 94 – 98, 2013.
- [11] J.-H. Jourdan, F. Pottier, and X. Leroy, "Validating LR (1) parsers," in *Programming Languages and Systems*. Springer, 2012, pp. 397–416.
- [12] N. Swamy, C. Hrițcu, and C. Keller, "Dependent Types and Multi-monadic Effects in F\*," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 2016. New York, NY, USA: ACM, 2016, pp. 256–270. [Online]. Available: <http://doi.acm.org/10.1145/2837614.2837655>
- [13] The Dafny project. [Online]. Available: <http://research.microsoft.com/en-us/projects/dafny/>
- [14] The WhyML project. [Online]. Available: <http://why3.lri.fr/>
- [15] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.
- [16] F\* tutorial. [Online]. Available: <https://www.fstar-lang.org/tutorial/>
- [17] Programming language F#. [Online]. Available: <http://fsharp.org/>
- [18] D. Firsov and T. Uustalu, "Certified normalization of context-free grammars," in *Proceedings of the 2015 Conference on Certified Programs and Proofs*. ACM, 2015, pp. 167–174.

# Performance Testing of Automated Theorem Provers Based on Sudoku Puzzle

Maxim Sabyanin, Dmitry Senotov, Grigory Skvortsov, Rostislav Yavorsky

Faculty of Computer Science

Higher School of Economics

Moscow, Russia

[Sabyanin.mx@gmail.com](mailto:Sabyanin.mx@gmail.com), [disenotov@gmail.com](mailto:disenotov@gmail.com),

[skvortsovg@yandex.ru](mailto:skvortsovg@yandex.ru), [ryavorsky@hse.ru](mailto:ryavorsky@hse.ru)

**Abstract**— This paper reports on work in progress on developing a test suite to assess performance of automated theorem provers Isabelle, Yices, and Z3. The developed tests are based on logical formalization of well-known Sudoku puzzle.

**Keywords**—automated theorem provers, testing, SAT solvers, performance testing, formal verification

## I. INTRODUCTION

In this paper we suggest a simple and user friendly way to test performance of different automates proof systems. The idea is to start from a well-known puzzle game Sudoku, reformulate it in logical terms and then use the obtained formula to assess performance of formal verification tools.

In our work we are interested to compare performance of the following tools: Z3, Isabelle, and Yices. Since not all these systems participate in regular competitions like SMT-COMP (see [1]), we were motivated to design our own tests.

## II. AUTOMATED THEOREM PROVERS

Theorem provers are widely used as core engines of formal verifications systems [2]. The following three are among the leaders, although they have slightly different purposes and application domains.

### A. Isabelle

Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols [4].

### B. Yices

Yices 2 is an SMT solver that decides the satisfiability of formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, bitvectors, scalar types, and tuples [3].

### C. Z3

Z3 is a state-of-the art theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over one or more theories. Z3 offers a compelling match for software analysis and verification tools, since several common software constructs map directly into supported theories [5].

## III. SUDOKU LOGICAL FORMULA

### A. The Original Puzzle

Sudoku is a very popular puzzle game and its goal is to fill a 9\*9 grid with numbers under certain rules. These rules are: each row, column and 3\*3 square must contain all of the digits from 1 to 9.

### B. Formalization

There are at least two ways to reformulate a Sudoku puzzle in logical terms. The first one is to use Boolean logic, that is to create a Boolean formula with 243 variables

$$\{ P_{ijk} \mid 1 \leq i \leq 9, 1 \leq j \leq 9, 1 \leq k \leq 9 \}$$

where  $P_{ijk}$  stands for a fact that cell  $(i, j)$  is filled with value  $k$ . The logical formula itself implements the rules of Sudoku.

Another way is to formalize the same rules using integer variables and arithmetical relations.

Below is a fragment of such a formula in Isabelle syntax:

```
valid e1 e2 e3 e4 e5 e6 e7 e8 e9 ==
```

```
(e1 \<noteq> e2) \<and> (e1 \<noteq> e3) \<and> (e1  
\<noteq> e4) \<and> (e1 \<noteq> e5) \<and> (e1  
\<noteq> e6) \<and> (e1 \<noteq> e7) \<and> (e1  
\<noteq> e8) \<and> (e1 \<noteq> e9) \<and> (e2  
\<noteq> e3) \<and> (e2 \<noteq> e4) \<and> (e2  
\<noteq> e5) \<and> (e2 \<noteq> e6) \<and> (e2  
\<noteq> e7) \<and> (e2 \<noteq> e8) \<and> (e2  
\<noteq> e9) \<and> (e3 \<noteq> e4) \<and> (e3  
\<noteq> e5) \<and> (e3 \<noteq> e6) \<and> (e3  
\<noteq> e7) \<and> (e3 \<noteq> e8) \<and> (e3  
\<noteq> e9) \<and> (e4 \<noteq> e5) \<and> (e4
```

```

\<noteq> e6) \<and> (e4 \<noteq> e7) \<and> (e4
\<noteq> e8) \<and> (e4 \<noteq> e9) \<and> (e5
\<noteq> e6) \<and> (e5 \<noteq> e7) \<and> (e5
\<noteq> e8) \<and> (e5 \<noteq> e9) \<and> (e6
\<noteq> e7) \<and> (e6 \<noteq> e8) \<and> (e6
\<noteq> e9) \<and> (e7 \<noteq> e8) \<and> (e7
\<noteq> e9) \<and> (e8 \<noteq> e9) \<and> (e1 < 10)
\<and> (e2 < 10) \<and> (e3 < 10) \<and> (e4 < 10) \<and>
(e5 < 10) \<and> (e6 < 10) \<and> (e7 < 10) \<and> (e8 <
10) \<and> (e9 < 10) \<and> (e1 \<ge> 1) \<and> (e2
\<ge> 1) \<and> (e3 \<ge> 1) \<and> (e4 \<ge> 1) \<and>
(e5 \<ge> 1) \<and> (e6 \<ge> 1) \<and> (e7 \<ge> 1)
\<and> (e8 \<ge> 1) \<and> (e9 \<ge> 1)

```

For Z3 besides others we used the following method to call the prover API:

```

void add_inequalities(z3::solver &solver, Sudoku& sudoku)
{
    for (int number_sq = 0; number_sq < 9; ++number_sq) {
        int start_i = (number_sq / 3) * 3,
            start_j = (number_sq % 3) * 3;

        for (int first_var = 0; first_var < 9; first_var++) {
            int i = start_i + dx[first_var],
                j = start_j + dy[first_var];
            if (sudoku[i][j] == EMPTY_CELL) {
                //continue;
            }

            for (int second_var = first_var + 1; second_var < 9;
                second_var++) {
                int a = start_i + dx[second_var],
                    b = start_j + dy[second_var];
                solver.add(sudoku.get_const({a, b}) !=
                    sudoku.get_const({i, j}));
            }
        }
    }
}

```

For Yices we used the Yices 2 language to formalize the Sudoku rules:

```

(assert (distinct e11 e12 e13 e14 e15 e16 e17 e18 e19 ))
(assert (distinct e21 e22 e23 e24 e25 e26 e27 e28 e29 ))
(assert (distinct e31 e32 e33 e34 e35 e36 e37 e38 e39 ))
(assert (distinct e41 e42 e43 e44 e45 e46 e47 e48 e49 ))
(assert (distinct e51 e52 e53 e54 e55 e56 e57 e58 e59 ))
(assert (distinct e61 e62 e63 e64 e65 e66 e67 e68 e69 ))
(assert (distinct e71 e72 e73 e74 e75 e76 e77 e78 e79 ))
(assert (distinct e81 e82 e83 e84 e85 e86 e87 e88 e89 ))
(assert (distinct e91 e92 e93 e94 e95 e96 e97 e98 e99 ))
(assert (distinct e11 e21 e31 e41 e51 e61 e71 e81 e91 ))

```

```

(assert (distinct e12 e22 e32 e42 e52 e62 e72 e82 e92 ))
(assert (distinct e13 e23 e33 e43 e53 e63 e73 e83 e93 ))
(assert (distinct e14 e24 e34 e44 e54 e64 e74 e84 e94 ))
(assert (distinct e15 e25 e35 e45 e55 e65 e75 e85 e95 ))
(assert (distinct e16 e26 e36 e46 e56 e66 e76 e86 e96 ))
(assert (distinct e17 e27 e37 e47 e57 e67 e77 e87 e97 ))
(assert (distinct e18 e28 e38 e48 e58 e68 e78 e88 e98 ))
(assert (distinct e19 e29 e39 e49 e59 e69 e79 e89 e99 ))
(assert (distinct e11 e12 e13 e21 e22 e23 e31 e32 e33 ))
(assert (distinct e41 e42 e43 e51 e52 e53 e61 e62 e63 ))
(assert (distinct e71 e72 e73 e81 e82 e83 e91 e92 e93 ))
(assert (distinct e14 e15 e16 e24 e25 e26 e34 e35 e36 ))
(assert (distinct e44 e45 e46 e54 e55 e56 e64 e65 e66 ))
(assert (distinct e74 e75 e76 e84 e85 e86 e94 e95 e96 ))
(assert (distinct e17 e18 e19 e27 e28 e29 e37 e38 e39 ))
(assert (distinct e47 e48 e49 e57 e58 e59 e67 e68 e69 ))
(assert (distinct e77 e78 e79 e87 e88 e89 e97 e98 e99 ))

```

Distinct is an operator that generalizes disequality. In other words, (distinct  $t_1 t_2 \dots t_n$ ) is true if  $t_1 \dots t_n$  are different from each other.

#### IV. THE RESULTS

The work is still in progress, yet we already have some figures. See figures 1 and 2 below. We started from a single puzzle, which has 22 predefined cells out of 81. According to the Sudoku rules the puzzle has a unique solution. Then we sequentially erased predefined and passed the task to prover under the test.

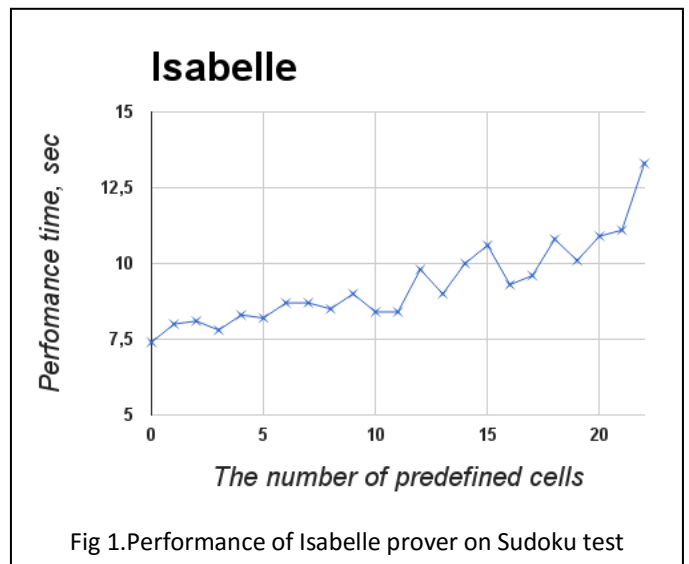


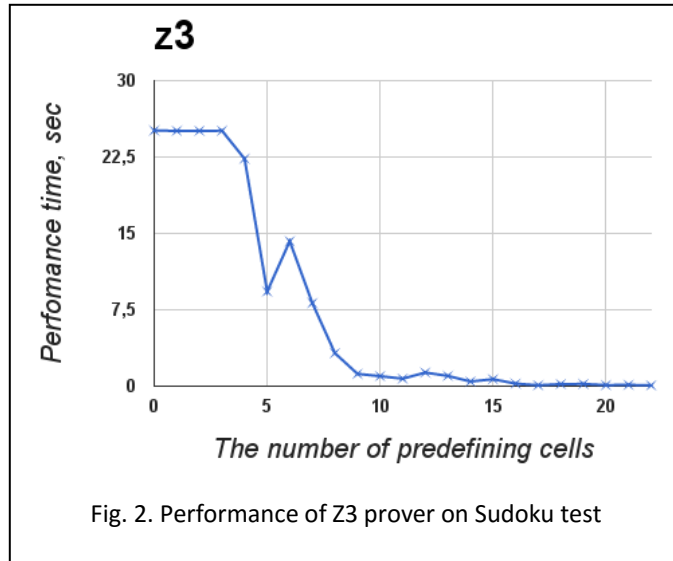
Fig 1. Performance of Isabelle prover on Sudoku test



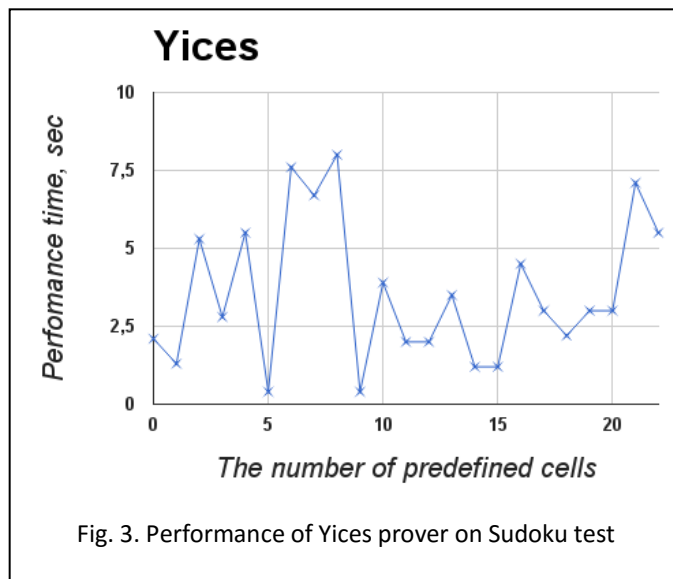
On the graphs the horizontal scale corresponds to the number of predefined cells, the vertical one is to show the time the prover took to solve the task. The first value, 0 on the horizontal scale, corresponds to the situation with most freedom, when no cell is predefined (so called zero-test).

Results for Isabelle are on Fig. 1 above.

The following graph is for Z3:



And the next graph is for Yices:



In addition, we used some puzzles to test Yices and Isabelle. For instance, “The Mephram diabolical Sudoku puzzle” from [6].

	9		7			8	6	
	3	1			5		2	
8		6						
		7		5				6
			3		7			
5				1		7		
						1		9
	2		6			3	5	
	5	4			8		7	

Fig 4. The Mephram diabolical Sudoku puzzle

It took Yices about 1,5 seconds to solve this puzzle. In comparison, Isabelle solved it in 12,2 seconds.

Second puzzle we tested Yices in is “Will Shortz’s puzzle 301”

	3	9	5					
			8				7	
				1		9		4
1			4					3
		7				8	6	
		6	7		8	2		
	1			9				5
					1			8

Fig 5. Will Shortz’s puzzle 301

It took Yices about 5 seconds to solve this puzzle, and it took Isabelle 10,8 seconds to do so.

## V. CONCLUSION

One can clearly see that the graph shapes are very different for different provers. Z3 turned out to be levels of magnitude more efficient on formulas with very little number of solutions. For example, it took approximately 76 milliseconds to solve the original puzzle. To compare, Isabelle worked on it 13.3 seconds, which is 175 times slower.

On the other hand, Isabelle outperformed Z3 on tasks which are not that strict, 7.4 seconds for Isabelle versus 25.1 seconds for Z3 for the zero-test.

As for the Yices, for these tests it always works faster than Isabelle, and it works faster than Z3 for relatively small amount of filled cells.

Such results were quite surprising for us. A possible explanation could be that the provers use different heuristics, which are targeted at different kinds of formulas.

## REFERENCES

- [1] Barrett, Clark, Leonardo De Moura, and Aaron Stump. "SMT-COMP: Satisfiability modulo theories competition." Computer Aided Verification. Springer Berlin Heidelberg, 2005.
- [2] Milne, George J. Formal specification and verification of digital systems. McGraw-Hill, Inc., 1993.
- [3] The Yices SMT Solver Overview, URL : <http://yices.csl.sri.com/>
- [4] Isabelle Overview, URL : <https://isabelle.in.tum.de/overview.html>
- [5] Z3 – a Tutorial, URL : <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.225.8231>
- [6] J. F. Crook, A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles. Notices of the ASM, Vol 57, No 4, April 2009  
<http://www.ams.org/notices/200904/rtx090400460p.pdf>



# Translation of Nested Petri Nets into Petri Nets for Unfoldings Verification

Vera Ermakova  
National Research University  
Higher School of Economics (HSE)  
Email: voermakova@edu.hse.ru

Irina Lomazova  
National Research University  
Higher School of Economics (HSE)  
Email: ilomazova@hse.ru

**Abstract**—Nested Petri nets (NP-nets) is an extension of the Petri nets formalism within the nets-within-nets approach, allowing to model systems of interacting dynamic agents in a natural way. One of the main problems in verifying of such systems is the State Explosion Problem. To tackle this problem for highly concurrent systems the unfolding method has proved to be very helpful. The purpose of this research is to study the application of unfoldings in the context of nested Petri nets and compare unfolding of NP-net translated into classical Petri net with direct component-wise unfolding.

**Keywords** —multi-agent systems; verification; Petri nets; nested Petri nets; unfoldings

## I. INTRODUCTION

Multi-agent systems have been studied explicitly for the last decades and can be regarded as one of the most advanced research and development area in computer science today. They are used in various practical fields and areas, such as artificial intelligence, cloud services, grid systems, augmented reality systems with interactive environment objects, information gathering, mobile agent cooperation, sensor information and communication.

However, multi-agent systems are very complex because of their distributed structure. They consist of interacting agents inhabiting an environment and having an autonomous behavior. When develop such a system, it is very important to check whether this system meets a given specification. That is why we create model for it: to check whether model works properly and only then we implement it. In order to do this check, verification approach is used.

One of the formalisms, successfully representing distributed systems behavior, is Petri nets. However, due to the flat structure of classical Petri nets, they are not so good for modeling complex multi-agent systems. For such systems a special extension of Petri nets, called nested Petri nets [1], can be used. Nested Petri nets naturally represent multi-agent systems structure because tokens in the main system net are Petri nets themselves, and can have their own behavior.

To check nested Petri net model properties one of the most common ways of verification, *model checking*, is used. The basic idea of model checking is to build a reachability (transition) graph and check properties on this graph. However, there is a crucial problem for verification of highly concurrent systems using model checking approach - a large number of

interleavings of concurrent processes (possible events sequencings in the system). This leads to the so-called state-space explosion problem. It means that if we add a single component to the system, the number of states will grow vastly.

To tackle this problem, unfolding theory [2], [3] was introduced. In [4] applicability of unfoldings for nested Petri nets is shown and the branching process of conservative nested Petri nets is defined in a component-wise manner. It was proved, that nested Petri nets satisfy the unfoldings fundamental property, and thus can be used for verification of conservative nested Petri nets same as classical unfoldings methods. A special subclass of nested Petri nets was considered here - *conservative safe nested Petri nets*. This means that net tokens, representing agents, cannot be destroyed or created, but can change the location in the system net and can change their inner states. Thus, the number of agents is constant and each agent is presented in a single copy in the system.

However, for this particular class of nested Petri nets it is possible to build equivalent classical Petri nets. Thus we are interested is unfolding approach proposed in [4] better in terms of verification complexity. To answer this question, we proposed to compare two ways of unfoldings: to make an unfolding directly for a nested Petri net, as it was proposed in [4], or to translate a nested Petri net (*source net*) into a classical Petri net (*target net*) and only then to build an unfolding for it.

a) *Related work*: Nested Petri nets (NP-nets) are widely used in modeling of distributed systems [5], [6], [7], serial or reconfigurable systems [8], [9], [10], protocol verification [11], coordination of sensor networks with mobile agents [12], innovative space system architectures [13], grid computing [14].

Several methods for NP-nets behavioral analysis were proposed in the literature, among them compositional methods for checking boundedness and liveness for nested Petri nets [15], translation of NP-nets into Colored Petri nets in order to verify them with CPNtools [16], verification of a subclass of recursive NP-nets with SPIN [17].

Unfolding approach and state-space explosion problem are explicitly studied in the literature. The original development in *unfoldings* (of P/T-nets) is due to [18]. McMillan [2] was the first to use unfoldings for verification. He introduced the concept of *complete finite prefixes* of unfoldings, and demon-

strated the applicability of this approach to the verification of asynchronous circuits.

The original McMillan's algorithm was used to solve the *executability* problem — to check whether a given transition can fire in the net. This algorithm can be used also for checking deadlock-freedom and for solving some other problems. Later, numerous improvements to the algorithm have been proposed ([19], [20], [21] to name a few); and the approach has been applied to high-level Petri nets [22], process algebras [23] and M-nets [22].

The general method for truncating unfoldings, which abstracts from the information one wants to preserve in the finite prefix of the unfolding, was proposed in [24], [25]. This method is based on the notion of a *cutting context*. We use this approach for defining branching processes and unfoldings of conservative nested Petri nets.

b) *The paper is organized as follows:* In Section II we present the basic notions of Petri nets and nested Petri nets. In Section III we present an algorithm for nested Petri nets into classical Petri nets translation. In Section IV we provide a comparison of two unfolding methods. Lastly, we discuss the applicability of our construction to the verification algorithms based on the canonical prefixes of unfoldings and classical unfoldings method on Petri nets.

## II. PRELIMINARIES

**Multisets.** Let  $S$  be a finite set. A *multiset*  $m$  over a set  $S$  is a function  $m : S \rightarrow \text{Nat}$ , where  $\text{Nat}$  is the set of natural numbers (including zero), in other words, a multiset may contain several copies of the same element.

For two multisets  $m, m'$  we write  $m \subseteq m'$  iff  $\forall s \in S : m(s) \leq m'(s)$  (the inclusion relation). The sum and the union of two multisets  $m$  and  $m'$  are defined as usual:  $\forall s \in S : (m + m')(s) = m(s) + m'(s)$ ,  $(m \cup m')(s) = \max(m(s), m'(s))$ .

### A. P/T-nets

Let  $P$  and  $T$  be two finite disjoint sets of *places* and *transitions* and let  $F \subseteq (P \times T) \cup (T \times P)$  be a *flow relation*. Then  $N = (P, T, F)$  is called a *P/T-net*.

A *marking* in a P/T-net  $N = (P, T, F)$  is a multiset over the set of places  $P$ . By  $\mathcal{M}(N)$  we denote a set of all markings in  $N$ . A *marked P/T-net*  $(N, M_0)$  is a P/T-net together with its *initial marking*  $M_0$ .

Pictorially,  $P$ -elements are represented by circles,  $T$ -elements by boxes, and the flow relation  $F$  by directed arcs. Places may carry tokens represented by filled circles. A current marking  $m$  is designated by putting  $m(p)$  tokens into each place  $p \in P$ .

For a transition  $t \in T$ , an arc  $(x, t)$  is called an *input arc*, and an arc  $(t, x)$  — an *output arc*. For each node  $x \in P \cup T$ , we define the *pre-set* as  $\bullet x = \{y \mid (y, x) \in F\}$  and the *post-set* as  $x \bullet = \{y \mid (x, y) \in F\}$ .

We say that a transition  $t$  in P/T-net  $N = (P, T, F)$  is *enabled* at a marking  $M$  if  $\bullet t \subseteq M$ . An enabled transition may *fire*, yielding a new marking  $M' = M - \bullet t + t \bullet$  (denoted  $M \xrightarrow{t} M'$ ). A marking  $M$  is called *reachable* if there exists

a (possibly empty) sequence of firings  $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \rightarrow \dots \rightarrow M$  from the initial marking to  $M$ . By  $\mathcal{RM}(N)$  we denote the set of all reachable markings in  $N$ .

A marking  $M$  is called *safe* iff for all places  $p \in P$  we have  $M(p) \leq 1$ . A marked P/T-net  $N$  is called *safe* iff every reachable marking  $M \in \mathcal{RM}(N)$  is safe. A *reachability graph* of a P/T-net  $(N, M_0)$  presents detailed information about the net behavior. It is a labeled directed graph, where vertices are reachable markings in  $(N, M_0)$ , and an arc labeled by a transition  $t$  leads from a vertex  $v$ , corresponding to a marking  $M$ , to a vertex  $v'$ , corresponding to a marking  $M'$  iff  $M \xrightarrow{t} M'$  in  $N$ .

### B. Classical Petri nets unfoldings

#### a) Branching processes and unfoldings of P/T-nets:

Unfoldings are used to define non-sequential (true concurrent) semantics of P/T-nets, and complete prefixes of unfoldings are used for verification. Here we give necessary basic notions and definitions, connected with unfoldings. Further details can be found in [26], [27].

Let  $N = (P, T, F)$  be a P/T-net. The following relations are defined on the set  $P \cup T$  of nodes in  $N$ :

- 1) the *causality* relation, denoted as  $<$ , is the transitive closure of  $F$ , and  $\leq$  is the reflexive closure of  $<$ ; if  $x < y$ , we say that  $y$  causally depends on  $x$ .
- 2) the *conflict* relation, denoted as  $\#$ : for nodes  $x, y \in P \cup T$ ,  $x \# y := \exists t, t' \in T. t \neq t' \wedge \bullet t \cap \bullet t' \neq \emptyset \wedge t \leq x \wedge t' \leq y$ ;
- 3) the *concurrency* relation, denoted as  $co$ : two nodes are *concurrent* if they are not in conflict and neither of them causally depends on the other.

For a set  $B$  of nodes we write  $co(B)$  iff all nodes in  $B$  are pairwise concurrent.

An *occurrence net* is a safe P/T-net  $ON = (B, E, G)$  s.t.

- 1)  $ON$  is acyclic;
  - 2)  $\forall p \in B : |\bullet p| \leq 1$ ;
  - 3)  $\forall x \in B \cup E$  the set  $\{y \mid y < x\}$  is finite, i.e., each node in  $ON$  has a finite set of predecessors;
  - 4)  $\forall x \in B \cup E : \neg(x \# x)$ , i.e., no node is in self-conflict.
- In occurrence nets, elements from  $B$  are usually called *conditions* and elements from  $E$  are called *events*.

As occurrence nets are represented using true concurrency semantics, we should clearly distinguish between them and interleaving concurrency. True concurrency is contrary to interleaving concurrency. It cannot be reduced to interleaving. The main difference between them is that in interleaved concurrency only one atomic action may happen. In contrast, in true concurrency there can be more than one atomic action. For example, in interleaving concurrency only one message can be sent from a server to a client in each step (moment of time).

From the observers point of view, true concurrency and interleaved concurrency behave the same. Also, interleaving concurrency is easier to handle in proofs, it is more reasonable in some problems to use simpler interleaving based concurrency (e.g. CCS and  $\pi$ -calculus). And it is a good true concurrency decomposition. However, when we deal with

timed computation, the difference between them becomes observable.

A *configuration*  $C$  in an occurrence net  $ON = (B, E, G)$  is a non-conflicting subset of nodes, which is downwards-closed under  $<$ , i.e.,  $\forall x, y \in C: \neg(x \# y)$ , and  $(x < y) \wedge y \in C$  implies  $x \in C$ . For each  $x \in B \cup E$  we define a *local configuration* of  $x$  to be  $[x] = \{y \mid y \in B \cup E, y < x\}$ . The definition of a local configuration can be straightforwardly generalized to any non-conflicting set of nodes  $X \subseteq B \cup E$ , namely  $[X] = \{y \mid y \in B \cup E, x \in X, y < x\}$ .

We define the set of branching processes of a given marked P/T-net  $N = (P, T, F, M_0)$  using the so-called *canonical representation*.

The set  $\mathcal{C}$  of *canonical names* for  $N$  is defined recursively to be the smallest set s.t. if  $x \in P \cup T$  and  $A$  is a finite subset of  $\mathcal{C}$ , then  $(A, x) \in \mathcal{C}$ .

A  $\mathcal{C}$ -Petri net is an occurrence net  $(B, E, G)$  such that:

- $B \cup E \subseteq \mathcal{C}$ ;
- $\forall (A, x) \in B \cup E, \bullet(A, x) = A$ .

The initial marking of a  $\mathcal{C}$ -Petri net is a subset of nodes  $\{(\emptyset, x) \mid (\emptyset, x) \in B\}$ . For each  $\mathcal{C}$ -Petri net  $CN$ , the morphism  $h$  maps the nodes of  $CN$  to the nodes of  $N$ :  $h((A, x)) = x$ . If  $h(y) = z$ , we say that  $y$  is labeled by  $z$ .

Let  $S$  be a (finite or infinite) set of  $\mathcal{C}$ -Petri nets. The union of  $S$  is defined component-wise, i.e.,

$$\bigcup S = (\bigcup_{(P,T,F,M) \in S} P, \bigcup_{(P,T,F,M) \in S} T, \bigcup_{(P,T,F,M) \in S} F, \bigcup_{(P,T,F,M) \in S} M).$$

The set of *branching processes* of a marked P/T-net  $N = (P, T, F, M_0)$  is defined as the smallest set satisfying the following conditions:

- 1) The occurrence net  $(I, \emptyset, \emptyset)$ , where  $I = \{(\emptyset, p) \mid p \in M_0\}$  (consisting of conditions  $I$  and having no events), is a branching process.
- 2) Let  $\mathcal{B}_1$  be a branching process and  $M$  be a reachable marking of  $\mathcal{B}_1$ , and  $M' \subseteq M$ , such that  $h(M') = \bullet t$  for some  $t$  in  $T$ . Let  $\mathcal{B}_2$  be a net obtained by adding an event  $(M', t)$  and conditions  $\{(\{(M', t)\}, p) \mid p \in t^\bullet\}$  to  $\mathcal{B}_1$ . Then  $\mathcal{B}_2$  is a branching process.
- 3) Let  $\mathcal{BB}$  be a (finite, or infinite) set of branching processes. The union  $\bigcup \mathcal{BB}$  is a branching process.

An example of a P/T-net and its branching process is shown in Figs. 1 and 2. The P/T-net  $PN_1$  has the initial marking  $\{p_1\}$  and is shown in Fig. 1. One of its possible branching processes is shown in Fig. 2, in which the labeling function  $h$  is indicated by labels on nodes.

A branching process  $\mathcal{B}_1 = ((P_1, E_1, F_1), h_1)$  is called a *prefix* of a branching process  $\mathcal{B}_2 = ((P_2, E_2, F_2), h_2)$  (denoted  $\mathcal{B}_1 \sqsubseteq \mathcal{B}_2$ ) iff  $P_1 \subseteq P_2$  and  $E_1 \subseteq E_2$ .

The maximal branching process of a net  $N$  w.r.t the prefix relation  $\sqsubseteq$  is called the *unfolding* of  $N$  and is denoted by  $U(N)$ .

The *fundamental property of P/T-nets unfoldings* [27] states that the behavior of the unfolding is equivalent to the behavior of the original net. Formally it can be formulated as follows.

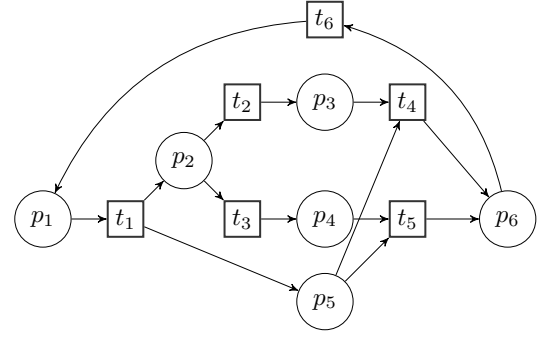


Figure 1. Petri net  $PN_1$

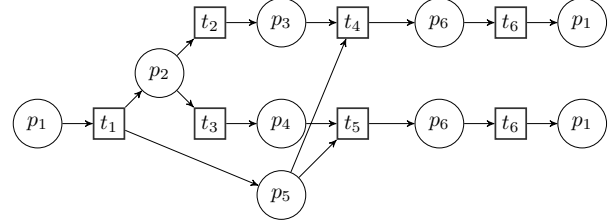


Figure 2. Branching process of  $PN_1$

*b) Fundamental property of P/T-nets unfoldings:* Let  $M$  be a reachable marking in a P/T-net  $N$ , and let  $M_U$  be a reachable marking in  $U(N)$  s.t.  $h(M_U) = M$ . Then

- 1) if there is a step  $M_U \xrightarrow{t_U} M'_U$  of  $U(N)$ , then there is a step  $M \xrightarrow{t} M'$  of  $N$ , such that  $h(t_U) = t \wedge h(M'_U) = M'$ ;
- 2) if there is a step  $M \xrightarrow{t} M'$  of  $N$ , then there is a step  $M_U \xrightarrow{t_U} M'_U$  in  $U(N)$ , such that  $h(t_U) = t \wedge h(M'_U) = M'$ .

In other words, the fundamental property of unfoldings states that the reachability graph of the unfolding is isomorphic to the reachability graph of the P/T-net. This property is crucial for the use of unfoldings in semantic study and verification.

Unfoldings were defined and studied for different classes of Petri nets, namely for high-level Petri nets [22], contextual nets [28], time Petri nets [29], Hypernets [30] (to name a few). All these constructions have similar properties, which act as a “sanity check”. Further in the paper we define an unfolding operation for nested Petri nets, which possesses a similar fundamental property.

### C. Nested Petri nets

In this paper we deal with nested Petri nets (NP-nets) — in particular, a proper subclass of NP-nets called *strictly conservative* NP-nets. The basic definition of nested Petri nets can be found in [1], [7]. Here we give a reduced definition, sufficient for defining conservative NP-nets.

In *nested Petri nets* (NP-nets), tokens may be Petri nets themselves. An NP-net consists of a *system net* and *element nets*. We call these nets the NP-net *components*. Marked element nets are *net tokens*. Net tokens, as well as usual black dot tokens, may reside in places of the system net.

Some transitions in NP-net components may be labeled with *synchronization labels*. Unlabeled transitions in NP-net components may fire autonomously, according to the usual rules for Petri nets. Labeled transitions in the system net should synchronize with transitions (labeled by the same label) in net tokens involved in this transition firing.

In strictly conservative NP-nets, net tokens cannot emerge or disappear. They can “move” from one place in a system net to another and “change” their marking, i.e., inner state. In the basic NP-net formalism new net tokens may be created, copied and removed as usual Petri net tokens. It should be noted that although this restriction is rather strong, many interesting multi-agent systems can be modeled with conservative NP-nets. Here we consider *safe* and *typed* NP-nets, i.e., each place in a system net can contain no more than one token: either a black dot token, or a net token of a specific type.

Let *Type* be a set of types, *Var* — a set of typed (over *Type*) variables, and *Lab* — a set of labels. A (typed) *nested Petri net* (NP-net) *NP* is a tuple  $(SN, (EN_1, \dots, EN_k), \nu, \lambda, W)$ , where

- $SN = (P_{SN}, T_{SN}, F_{SN})$  is a P/T net called a *system net*;
- for  $i = \overline{1, k}$ ,  $EN_i = (P_{EN_i}, T_{EN_i}, F_{EN_i})$  is a P/T net called an *element net*, where all sets of places and transitions in the system and element nets are pairwise disjoint; we suppose, each element net is assigned a type from *Type*;
- $\nu : P_{SN} \rightarrow Type \cup \{\bullet\}$  is a *place-typing function*;
- $\lambda : T_{NP} \rightarrow Lab$  is a partial transition labeling function, where  $T_{NP} = T_{SN} \cup T_{EN_1} \cup \dots \cup T_{EN_k}$ ; we write that  $\lambda(t) = \perp$  when  $\lambda$  is undefined at  $t$ .
- $W : F_{SN} \rightarrow Var \cup \{\bullet\}$  is an *arc labeling function* s.t. for an arc  $r$  adjacent to a place  $p$  the type of  $W(r)$  coincides with the type of  $p$ .

A *binding* of  $t$  is a function  $b$  assigning a value  $b(v)$  (of the corresponding type) from  $A$  to each variable  $v$  occurring in  $W(t)$ . A marked element net is called a *net token*. In what follows for a given NP-net by  $A_{net} = \{(EN, m) \mid \exists i = \overline{1, \dots, k} : EN = EN_i, m \in \mathcal{M}(EN_i)\}$  we denote the set of all (possible) net tokens, and by  $A = A_{net} \cup \{\bullet\}$  the set of all net tokens extended with a black dot token.

A *system-autonomous step* is the firing of an unlabeled transition  $t \in T_{SN}$  in the system net according to the firing rule for high-level Petri nets (e.g., colored Petri nets [31]), as described above.

A *synchronization step*. Let  $t$  be a transition labeled  $\lambda$  in the system net  $SN$ , let  $t$  be enabled in a marking  $M$  w.r.t. a binding  $b$  and let  $\alpha_1, \dots, \alpha_n \in A_{net}$  be net tokens involved in this firing of  $t$ . Then  $t$  can fire provided that in each  $\alpha_i$  ( $1 \leq i \leq n$ ) a transition labeled by the same synchronization label  $\lambda$  is also enabled. The synchronization step goes then in two stages: first, firing of transitions in all net tokens involved in the firing of  $t$  and then, firing of  $t$  in the system net w.r.t. binding  $b$ . An NP-net  $NP$  is called *safe* iff in every reachable marking in  $NP$  there are not more than one token in each place in the system net, and not more than one token in each net token place. Hereinafter we consider only safe NP-nets.

#### D. Conservative NP-nets

Now we give a definition of (*strictly*) *conservative NP-nets*, as well as some related definitions. We then define an unfolding operation for a simple class of strictly conservative nets.

A safe NP-net  $N = (SN, (EN_1, \dots, EN_k), \nu, \lambda, W)$  is called *strictly conservative* iff

- 1) For each  $t \in T_{SN}$  and for each  $p \in \bullet t$ ,  $\exists! p' \in t \bullet$ .  $W(p, t) = W(t, p')$  or  $W(p, t) = \bullet$
- 2) For each  $t \in T_{SN}$  and for each  $p \in t \bullet$ ,  $\exists! p' \in \bullet t$ .  $W(p', t) = W(t, p)$  or  $W(p, t) = \bullet$

The definition of strict conservativeness ensures that no net token emerges or disappears after a transition firing in the system net.

Note that in [32] NP-nets are called conservative, iff tokens cannot disappear after a transition firing, but can be copied; hence, the number of net tokens in such conservative NP-nets can be unlimited. Here we consider a more restrictive subclass of NP-nets with a stable set of net tokens (tokens cannot be copied). Hereinafter we consider only strictly conservative NP-nets, and call them just conservative nets for short.

### III. TRANSLATION OF SAFE CONSERVATIVE NP-NETS INTO P/T-NETS

As reachability graph of the unfolding is isomorphic to the reachability graph of the P/T-net, unfoldings can be used in verification. Since safe conservative nested Petri nets have finite number of states, it will be apparent to assume, that they can be translated into classical Petri nets and then can be unfolded according to the classical unfolding rules for further verification.

To make a correct translation we have to set a number of requirements for a translation. The main goal for building a model is the possibility to make a simulation. Simulation implies behavioral equivalence: a possibility to repeat all possible moves of one model on another model. Behavioral equivalence is guaranteed by establishing strong bisimulation equivalence between states of two models. The second requirement is about constructing a reachability graph. It means that we need exact correspondence between nodes (states) of our model. If these two requirements are met, we can build a translation algorithm which allows us to use target model having the same behavioral properties like original for verification and analysis.

Now we present an algorithm for translating a conservative safe nested Petri net into a safe P/T net.

The algorithm will be illustrated by an example of a NP-net  $NP_2$ , shown in Fig. 3 (system net) and 4 (element nets). This net will be translated into a safe P/T net  $PN$ .

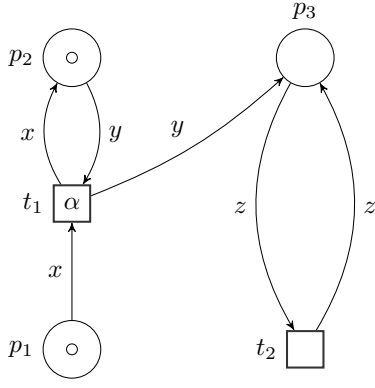


Figure 3. NP-net  $NP_2$

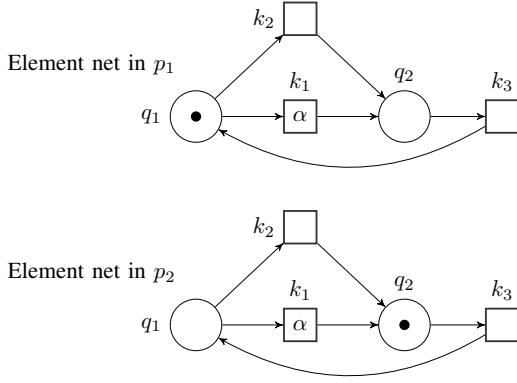


Figure 4. NP-net  $NP_2$

**The translation algorithm:** Let  $NP = (SN, (E_1, \dots, E_k), v, \lambda, W)$  be an NP-net with a set  $NTok$  of identified net tokens in the initial marking. By  $I$  we denote the set of all identifiers used in  $NTok$ , and by  $I_E \subseteq I$  the subset of all identifiers for net tokens of type  $E$ . The net  $NP$  will be translated into a P/T net  $PN = (P_{PN}, T_{PN}, F_{PN})$  with an initial marking  $m_0$ .

- 1) First, we define the set  $P_{PN}$  of places of the target net  $PN$ . For each type  $E$  of some place in the system net  $SN$  we create a set  $\mathfrak{S}_E$  of places for  $P_{PN}$ . The set  $\mathfrak{S}_E$  will contain a copy of each place of type  $E$  in the system net for each net token of type  $E$  (labeled by net token identifiers) and a copy of each place in  $P_E$  for each net token of type  $E$ , i.e.  $\mathfrak{S}_E = \{(p, id) | p \in P_{SN}, v(p) = E, id \in I_E\} \cup \{(q, id) | q \in P_E, id \in I_E\}$ . For a place  $p$  in  $SN$  with black token type we create just one copy of  $p$  without any identifier. Then the set  $P_{PN}$  of places for the target net  $PN$  is created as the union of all these sets. The first step is depicted in a Fig. 5.

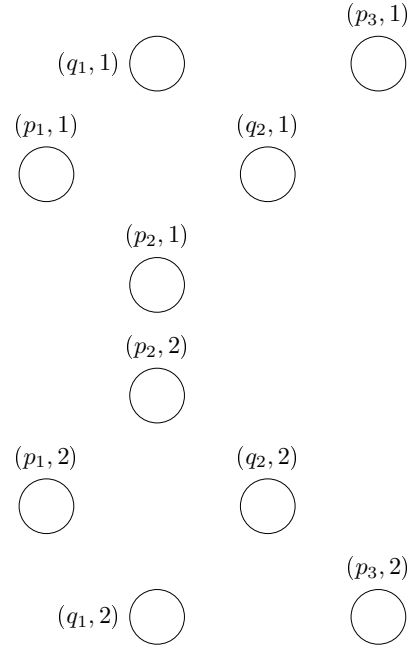


Figure 5. Creation of places for  $PN$

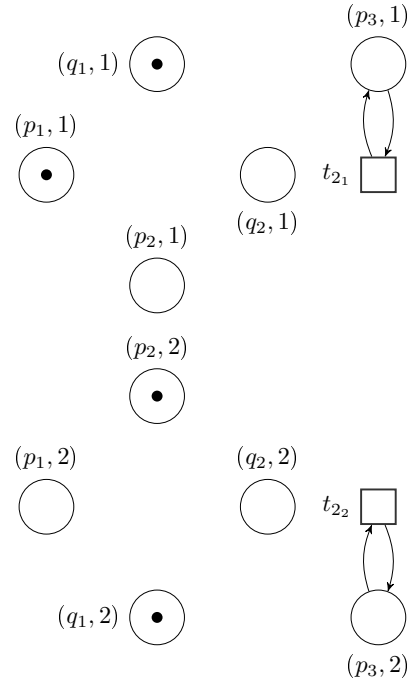


Figure 6. System-autonomous step

- 2) To define the initial marking for  $PN$  we define an encoding of markings on places from  $P_{NP}$  in a NP-net by markings on constructed places from  $P_{PN}$ . If a net token  $\eta = (id, E, m)$  resides in a place  $p$  in a marking  $M$  of the system net, then in the target net there are black tokens in the place  $(p, id)$ , and all places  $(q, id)$  for all  $q$  s.t.  $m(q) = 1$ . If a place of black token type in  $SN$  has a black token, then the only corresponding place

in  $PN$  is also marked by a black token. It is easy to see that this encoding defines a one-to-one correspondence between markings in a safe conservative NP-net and safe markings in  $PN$ .

In our example the first element net resides in a place  $p_1$ , second - in  $p_2$ . Thus, correspondingly, we define marking in a places  $(p_1, 1)$  and  $(p_2, 2)$ . The same way marking for places  $(q_1, 1)$  and  $(q_1, 2)$  is defined.

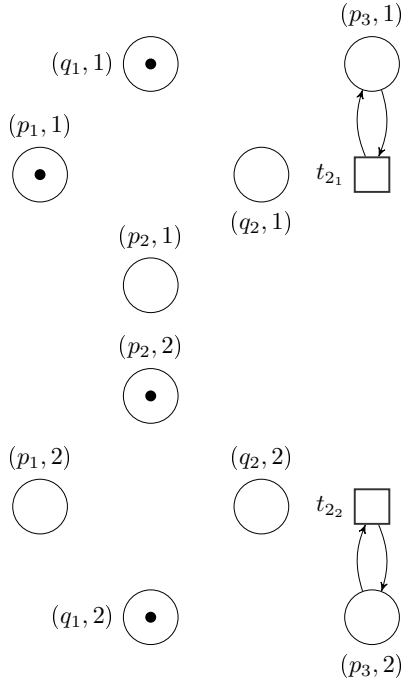


Figure 7. System-autonomous step

- 3) For each autonomous transition  $t$  in a system net  $SN$  we build a set  $\mathcal{T}_t$  of transitions as follows. Each input arc variable of  $t$  may be, generally speaking, be binded to any of identified net token of the corresponding type. So, for each such binding we construct a separate transition for  $PN$  with appropriate input and output arcs.

Thus for the transition  $t_2$  we construct two transitions:  $t_{2_1}$  and  $t_{2_2}$ . It is shown in Fig. 7.

- 4) For each autonomous transition in a net token from  $NTok$  identifies= $d$  with  $id$  we construct a similar transition on places labeled with  $id$ . Thus in our example net we obtain four transitions:  $k_{2_1}$ ,  $k_{2_2}$ ,  $k_{3_1}$  and  $k_{3_2}$ . Element-autonomous step is illustrated in Fig. 8.

- 5) A firing of a synchronization transition supposes simultaneous firing of a transition, which belongs to a system net, and firing of some transition, which has the same label in each involved net token. So synchronization step is a combination of Step 3 and Step 4. Thus as in our example there are two element nets, we add transitions for each net, marked with  $\alpha_1$  and  $\alpha_2$ . Suchwise we can model a synchronization step for every possible initial marking in a system net, which is shown in Fig. 9.

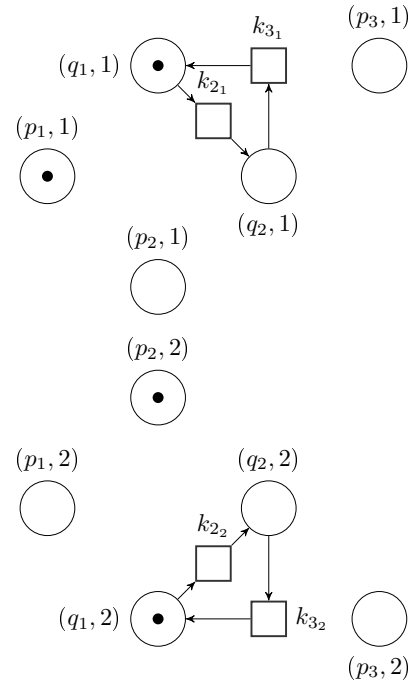


Figure 8. Element-autonomous step

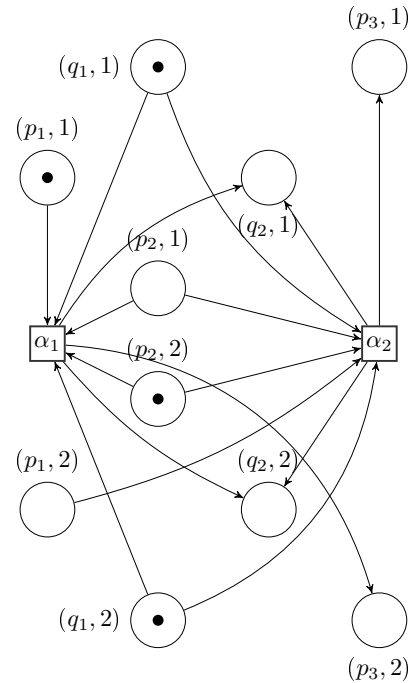


Figure 9. Synchronization step

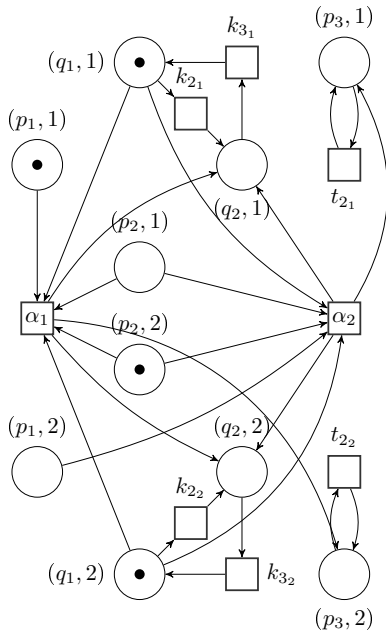


Figure 10. Translated net  $NP_2$

**Theorem.** Let  $NP$  be a NP-net. Let also  $PN$  be a P/T net, obtained from  $NP$  by the translation, described above. Then reachability graphs of  $NP$  and  $PN$  are isomorphic. Step 2 of the algorithm defines a one-to-one correspondence between reachable markings of nets  $NP$  and  $PN$ . It is easy to see that according to translation definition corresponding firing steps in both nets do not violate this correspondence.

#### IV. UNFOLDINGS

##### A. Branching Processes of a Conservative NP-net

A possible branching process of  $NP_2$  is shown in Fig. 11. In Fig. 11, a transition is labeled with  $t$ , if it is of the form  $(A, t)$ , and a place is labeled with  $(p, N)$  if it is of the form  $(A, p, N)$ .

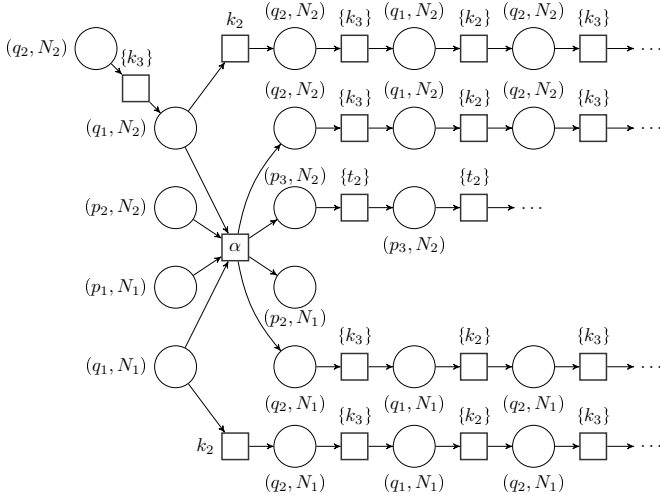


Figure 11. Branching process of  $NP_2$

A method for direct unfolding of nested Petri nets was proposed in [4]. It was shown that main properties of Petri net unfolding are valid for conservative safe Petri nets. Unfoldings for NP-net are defined using branching processes, similarly to the case of Petri nets. It allows us to avoid construction of the intermediate net. It is interesting to compare complexity of these two methods, the method proposed in [4] and the method based on nested Petri nets into Petri nets translation.

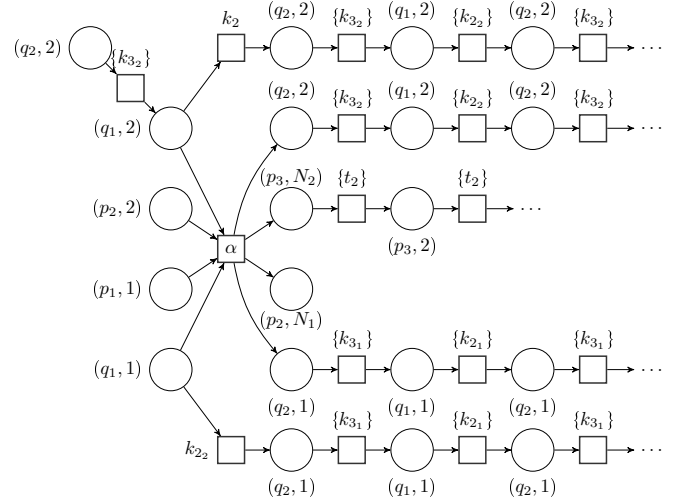


Figure 12. Branching process of translated net  $NP_2$

##### B. Comparing two ways of nested Petri net unfolding

We have shown that each conservative safe NP-net can be converted into a behaviorally equivalent classical Petri net, namely their reachability graphs are isomorphic. So, to construct unfoldings for NP-net we can either translate it into a P/T net and then apply P/T net unfolding, or directly apply to NP-nets unfoldings, as it is described in the previous subsection.

The fundamental property of unfoldings states that the reachability graph of the unfolding is isomorphic to the reachability graph of the initial net. Since the fundamental property holds both for P/T net unfoldings and for NP-net unfoldings, we can immediately conclude that both approaches give the same (up to isomorphism) result. We show it in Fig. 11, Fig. 12.

The difference is in the complexity of these two solutions. It is easy to see, that when there are several net tokens of the same type in the initial marking, the translation leads to a significant net growth. Thus e.g. for a system net transition with  $n$  input places of the same type and  $k$  tokens of this type in the initial marking we are to construct  $k^n$  copies of this transition in the target P/T net. And it is rather clear, that we cannot avoid this, since we are to distinguish markings of net tokens residing in different places, and hence to provide different P/T net transitions for different modes of system net transitions firings.



To check the advantage of the direct unfolding method w.r.t. time complexity for concrete examples we've developed a software application which allows

- 1) translation of a conservative safe nested Petri net into a P/T net and then building an unfolding for it;
- 2) building an unfolding directly for a nested Petri net.

We expected that a large number of net tokens cause significant net growth during translation. To get representative results, we conducted experiments on nets having similar structure, but different number of element nets with different types.

Even experiments with rather small models confirm our assumptions. Thus for our example net  $NP_2$  we've got 0.38 ms. for the direct unfolding, and 0.54 ms. for unfolding via the translation into a P/T net. So, even in the case of two net tokens we get a noticeable difference in time.

If we are dealing with a system, which consists of a large number of net tokens and incoming arcs, after applying translation of a nested Petri net into a P/T net the net graph will increase strongly. Since we do not know in advance, which transitions will be used in the unfolding, we should create an intermediate graph with a lot of transitions unnecessary for unfolding.

## V. CONCLUSION

In this paper we proposed the unfolding method, which is based on equivalent translation of NP-nets into safe P/T nets and then applying standard unfolding procedure described in the literature. We also compared it to existing direct unfolding method, proposed and justified in [4].

For that we've developed and justified an algorithm for translation of a safe conservative NP-net into an equivalent P/T net. Analysis of the algorithm complexity allows us to conclude that the direct unfolding has a distinct advantage in time complexity. To check this advantage with practical examples we've implemented the algorithms for translation and unfolding. Experiments on small nets have demonstrated the anticipated benefits of direct unfolding.

For further work, we plan to enlarge the complexity of nets and number of experiments.

## ACKNOWLEDGEMENT

This work is supported by the Basic Research Program at the National Research University Higher School of Economics and Russian Foundation for Basic Research, project No. 16-01-00546.

## REFERENCES

- [1] Lomazova, I.A.: Nested Petri nets—a formalism for specification and verification of multi-agent distributed systems. *Fundamenta Informaticae* **43**(1) (2000) 195–214
- [2] McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: *Computer Aided Verification*, Springer (1992) 164–177
- [3] Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, event structures and domains, part i. *Theoretical Computer Science* **13**(1) (1981) 85–108
- [4] Frumin, D., Lomazova, I.A.: Branching processes of conservative nested Petri nets. In: *VPT@ CAV*. (2014) 19–35
- [5] Lomazova, I.A., van Hee, K.M., Oanea, O., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Nested nets for adaptive systems. *Application and Theory of Petri Nets and Other Models of Concurrency*, LNCS (2006) 241–260
- [6] Lomazova, I.A.: Modeling dynamic objects in distributed systems with nested petri nets. *Fundamenta Informaticae* **51**(1-2) (2002) 121–133
- [7] Lomazova, I.A.: Nested petri nets for adaptive process modeling. In: *Pillars of computer science*. Springer (2008) 460–474
- [8] López-Mellado, E., Villanueva-Paredes, N., Almeyda-Canepa, H.: Modelling of batch production systems using Petri nets with dynamic tokens. *Mathematics and Computers in Simulation* **67**(6) (2005) 541–558
- [9] Kahloul, L., Djouani, K., Chaoui, A.: Formal study of reconfigurable manufacturing systems: A high level Petri nets based approach. In: *Industrial Applications of Holonic and Multi-Agent Systems*. Springer (2013) 106–117
- [10] Zhang, L., Rodrigues, B.: Nested coloured timed Petri nets for production configuration of product families. *International journal of production research* **48**(6) (2010) 1805–1833
- [11] Venero, M.L.F., da Silva, F.S.C.: Modeling and simulating interaction protocols using nested Petri nets. In: *Software Engineering and Formal Methods*. Springer (2013) 135–150
- [12] Chang, L., He, X., Lian, J., Shatz, S.: Applying a nested Petri net modeling paradigm to coordination of sensor networks with mobile agents. In: *Proc. of Workshop on Petri Nets and Distributed Systems*, Xian, China. (2008) 132–145
- [13] Cristini, F., Tessier, C.: Nets-within-nets to model innovative space system architectures. In: *Application and Theory of Petri Nets*. Springer (2012) 348–367
- [14] Mascheroni, M., Farina, F.: Nets-within-nets paradigm and grid computing. In: *Transactions on Petri Nets and Other Models of Concurrency V*. Springer (2012) 201–220
- [15] Dworżański, L.W., Lomazova, I.A.: On compositionality of boundedness and liveness for nested Petri nets. *Fundamenta Informaticae* **120**(3-4) (2012) 275–293
- [16] Dworżański, L., Lomazova, I.A.: Cpn tools-assisted simulation and verification of nested petri nets. *Automatic Control and Computer Sciences* **47**(7) (2013) 393–402
- [17] Venero, M.L.F.: Verifying cross-organizational workflows over multi-agent based environments. In: *Enterprise and Organizational Modeling and Simulation*. Springer (2014) 38–58
- [18] Winskel, G.: *Event structures*. Springer (1986)
- [19] Bonet, B., Haslum, P., Hickmott, S., Thiébaux, S.: Directed unfolding of petri nets. In: *Transactions on Petri Nets and Other Models of Concurrency I*. Springer (2008) 172–198
- [20] McMillan, K.L.: A technique of state space search based on unfolding. *Form. Methods Syst. Des.* **6**(1) (1995) 45–65
- [21] Heljanko, K.: Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. *Fundamenta Informaticae* **37**(3) (1999) 247–268
- [22] Khomenko, V., Koutny, M.: Branching processes of high-level Petri nets. In: Garavel, H., Hatcliff, J., eds.: *Tools and Algorithms for the Construction and Analysis of Systems*. Volume 2619 of *Lecture Notes in Computer Science*. Springer (2003) 458–472
- [23] Langerak, R., Brinksma, E.: A complete finite prefix for process algebra. In: *Computer Aided Verification*, Springer (1999) 184–195
- [24] Khomenko, V., Koutny, M., Vogler, W.: Canonical prefixes of Petri net unfoldings. *Acta Informatica* **40**(2) (2003) 95–118
- [25] Khomenko, V.: *Model Checking Based on Prefixes of Petri Net Unfoldings*. Ph.D. Thesis, School of Computing Science, Newcastle University (2003)
- [26] Esparza, J., Heljanko, K.: Unfoldings: a partial-order approach to model checking. Springer (2008)
- [27] Engelfriet, J.: Branching processes of Petri nets. *Acta Informatica* **28**(6) (1991) 575–591
- [28] Baldan, P., Corradini, A., Knig, B., Schwoon, S.: Mcmillans complete prefix for contextual nets. In: Jensen, K., Aalst, W.M., Billington, J., eds.: *Transactions on Petri Nets and Other Models of Concurrency I*. Volume 5100 of *Lecture Notes in Computer Science*. Springer (2008) 199–220
- [29] Fleischhack, H., Stehno, C.: Computing a finite prefix of a time Petri net. In: Esparza, J., Lakos, C., eds.: *Application and Theory of Petri Nets 2002*. Volume 2360 of *Lecture Notes in Computer Science*. Springer (2002) 163–181

- [30] Mascheroni, M.: Hypernets: a Class of Hierarchical Petri Nets. Ph.D. Thesis, Facolt di Scienze Naturali Fisiche e Naturali, Dipartimento di Informatica Sistemistica e Comunicazione, Università Degli Studi Di Milano Bicocca (2010)
- [31] Jensen, K., Kristensen, L.M.: Coloured Petri nets: modelling and validation of concurrent systems. Springer (2009)
- [32] Dworżański, L.W., Lomazova, I.A.: On compositionality of boundedness and liveness for nested Petri nets. *Fundamenta Informaticae* **120**(3) (2012) 275–293

# Automatic Code Generation from Nested Petri nets to Event-based Systems on the Telegram Platform

Denis Samokhvalov

National Research University Higher School of Economics  
disamokhvalov@edu.hse.ru

Leonid Dworzanski

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

**Abstract**—Nested Petri net formalisms is an extension of coloured Petri net formalism that uses Petri Nets as tokens. This formalism allows to create comprehensive models of multi-agent systems, simulate, verify and analyse them in a formal and rigorous way. Multi-agent systems are found in many different fields — from safety critical systems to everyday networks of personal computational devices. While several methods and tools were developed for modelling and analysis of NP-nets models, the automatic code-generation from NP-nets is still under active development.

In this paper, we demonstrate how Nested Petri net formalism could be applied to model operations coordination systems and automatically generate executable code for the Telegram platform. We augment the NP-nets models with annotations on the Action Language, which enables us to link transition firings to Telegram Bot API calls. The suggested approach is illustrated by the example of a search and rescue coordination system.

**Keywords**—*nested petri nets, telegram bot api, action language, event-based systems, code-generation.*

## I. INTRODUCTION

Messengers have become the integral part of our life in recent years; and, almost all the people who have Whatsapp, Viber or Telegram installed on their mobile devices use them in everyday life. That is all because of hands-on approach in terms of receiving and sending information. Telegram Bot API (TBA)[1] appeared not so long time ago has made a breakthrough in messengers evolution; and, many IT and business experts see the great potential in appliance of the tool for both business and computer science domains.

The variety of TBA usage shows the great diversity of different applied domains starting with service bots, which are designed in order to meet customers requirements, ending with Artificial Intelligence bots (e.g. YandexBot), which can answer different kind of queries and even strike up and sustain a coherent conversation. The one sphere where TBA could be applied in — people coordinating in different types of special operations. These operations turn out to be extremely difficult to plan and support when it comes to coordination of big squads; especially, in the state of emergency cases. A thorough planning of search and rescue or military operations are rather struggling to deal with, because of the lack of time to create a detailed schedule of part-taking for each agent and deprecated methods for sending and receiving notifications from agents who are involved in those operations. TBA provides a great opportunity for that purpose because it is extremely easy to use when the bot logic is designed according to a consecutive and well-structured scheme. However, it is not easy to create

a coherent TBA logic, because it requires programming skills and is time-consuming. As the time factor plays a crucial role, this makes such system much less attractive and unsuitable in the fast changing context of emergency and rescue operations.

Nested Petri Nets (NP-nets) are a well-known formalism which provides an approach for modelling multi-agent systems [2], [3], [4], [5]. NP-nets are generally used to describe the complex processes with dynamic hierarchical structure. NP-nets are convenient for specification of that kind of processes because of the visible and coherent structure [6]. A number of methods for the analysis and verification of NP-nets were developed [7], [8], [9]. However the practical application is impeded by the necessity of manual implementation of the constructed model. Even if the model correctness is verified, code defects can be introduced on the error-prone implementation phase of software construction process. The reasons for such defects: different understanding of the model by a software architect and software developers; the complex behaviour of multi-agent systems with dynamic structure; the distributed systems testing and debugging problems. The alternative to manual coding is automatic codegeneration from the model to an executable system. Automatic generation provide considerable saving of the project resources, reproducible quality of the generated code, better support for round-trip developing by regenerating code after model changes. The approach does not guarantee zero-defect implementation, but, after long term usage, a codegeneration system becomes reliable and allows to obtain code with reproducible quality.

The goal of the project is to develop a codegeneration system which allows to automatically construct multi-agent systems on the Telegram platform from NP-nets models. The generated software is designed according to the event-base paradigm and consists of a complex Telegram Bot and mobile Telegram applications. The main purpose of the Telegram bot is to coordinate and communicate with the agents according to the original NP-net model.

The section II contains basic notation and definitions. In the section III, a motivating example of Search and Rescue coordination system modelled with the NP-nets formalism is given. In the section IV, we provide the architecture and technical details on the implementation of the automatic code generation. The section V contains the suggested action language description. In the section VI, we discuss the application of the suggested technology to the motivating example. The section VII concerns the related work, the previous studies on NP-nets translations, and further directions.

## II. PRELIMINARIES

At first, we provide the classical definition of a Petri Net.

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

- $P$  and  $T$  are disjoint finite sets of *places* and *transitions*, respectively;
- $F \subseteq (P \times T) \cup (T \times P)$  is a set of *arcs*;
- $W : F \rightarrow \mathbb{N} \setminus \{0\}$  – an *arc multiplicity function*, that is, a function which assigns every arc a positive integer called an *arc multiplicity*.

A *marking* of a Petri net  $(P, T, F, W)$  is a multiset over  $P$ , i.e. a mapping  $M : P \rightarrow \mathbb{N}$ . By  $\mathcal{M}(N)$  we denote the set of all markings of the P/T-net  $N$ .

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

For simplicity, we consider here only two-level NP-nets, where net tokens are classical Petri nets.

**Definition 2.2:** A *nested Petri net* is a tuple  $NPN = (Atom, Expr, Lab, SN, (EN_1, \dots, EN_k))$  where

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

Let  $Con$  be a set of *constants* interpreted over  $A = A_{net} \cup \{\bullet\}$ ; and,  $A_{net} = \{(EN, m) \mid \exists i = 1, \dots, k : EN = EN_i, m \in \mathcal{M}(EN_i)\}$  is a set of marked element nets. Let  $Var$  be a set of *variables*. Then the expressions of *Expr* are multisets over  $Con \cup Var$ . The arc labelling function  $W$  is restricted such that: constants or multiple instances of the same variable are not allowed in input arc expressions of transitions; constants and variables in the output arc expressions correspond to the types of output places; and, each variable in an output arc expression of a transition occurs in one of the input arc expressions of the transition.

A marking  $M$  of an NP-net  $NPN$  is a function mapping each  $p \in P_{SN}$  to a multiset  $M(p)$  over  $A$ . The set of all markings of an NP-net  $NPN$  is denoted by  $\mathcal{M}(NPN)$ . Let

$Vars(e)$  denote a set of variables in an expression  $e \in Expr$ . For each  $t \in T_{SN}$  we define  $W(t) = \{W(x, y) \mid (x, y) \in F_{SN} \wedge (x = t \vee y = t)\}$  – all expressions labelling arcs incident to  $t$ . A *binding*  $b$  of a transition  $t$  is a function  $b : Vars(W(t)) \rightarrow A$ , mapping every variable in the  $t$ -incident arc expression to a token. We say that a transition  $t$  is *active* in a binding  $b$  iff  $\forall p \in \{p \mid (p, t) \in F_{SN}\}$ :  $b(W(p, t)) \subseteq M(p)$ . An active transition  $t$  may *fire* yielding a new marking  $M'(p) = M(p) - b(W(p, t)) + b(W(t, p))$  for each  $p \in P_{SN}$  (denoted as  $M \xrightarrow{t[b]} M'$ ).

A behaviour of an NP-net consists of three kinds of steps. A *system-autonomous step* is a firing of a transition, labelled with  $\tau$ , in the system net without changing the internal markings of the involved tokens. An *element-autonomous step* is a transition firing in one of the element nets according to the standard firing rules for P/T-nets. An *autonomous step* in a net token changes only this token inner marking. An *autonomous step* in a system net can move, copy, generate, or remove tokens involved in the step, but doesn't change their inner markings.

A (*vertical*) *synchronization step* is a simultaneous firing of a transition labelled with some  $\lambda \in Lab$  in a system net with firings of transitions labelled with the same  $\lambda$  in all net consumed tokens involved in this system net transition firing. For further details see [5]. Note, however, that here we consider a typed variant of NP-nets, when a type of an element net is instantiated to each place.

## III. MOTIVATING EXAMPLE

Search and rescue operations is what happens all over the world; they require the well-trained and skilled employees, well-structured planning, and knowledgeable human management. There were 2447 emergency callouts registered in Russia throughout 2005–2014 [10], and about 100 times more in USA [11]. Earthquakes, water floods, and hurricanes hit the earth rarely than ordinary emergency cases like fires or gas leaks, but they leave whole regions and even countries devastated, thousands of people killed or lost without a trace. Therefore, the crucial goal of rescuers is to treat such cases quickly and cohesively.

In this example we will explain how a particular search and rescue operation in an earthquake could be handled with a multi-agent model based on the nested Petri net formalism. First, we need to introduce the purposes of the basic components which we will use further to design our search and rescue coordination plan. Our model relies on two basic components:

- **System net** – the main component of an NP-net which is a high level Petri net. It will be used to define the activity coordination of the agents involved in the operation. The system net will be implemented on the Telegram platform to receive the notifications from agents and to process them with the Action Language (AL) event handlers assigned to the transitions of the system net;
- **Element net** – represents the activity of a particular agent type that is supposed to be performed by the agent while taking part in the operation. There are two element nets in our example. The first one corresponds

to the acting plan for medical workers involved in the operation, while the second one will provide the plan for the rescues participating in the operation.

The system net in Fig 1 represents the main model of our operation. Basically, it reflects the dependence of the agent actions on server responses. In other words, it describes how an operation coordinator interacts with the rescuers and medics and reacts on their signals to the server. The model deals only with those agent requests where coordinators answer is essential for the further operation progress. The actions happen when a particular agent reaches a state and the coordinator response expected are defined with AL code assigned to the system net transitions. To understand how the model works, we need to understand how the agents intercommunicate with the server coordinating this operation.

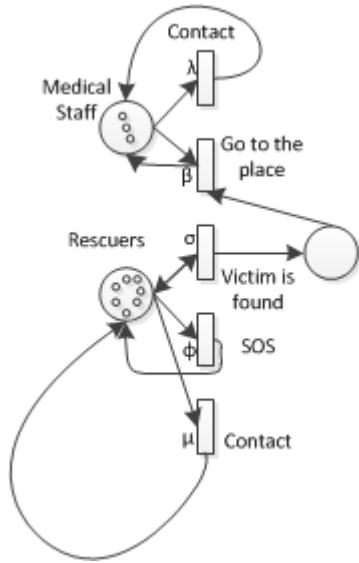


Fig. 1. The system net example.

In the initial marking of the places “Medical Staff” and “Rescuers” of the system net, there are all the agents – rescuers and medical staff respectively. The transitions have the next functions:

- Transitions T0 and T4 “**Contact**” — handle communication between the rescuers and the coordinator.
- Transition T1 “**Go to the place**” — represents the event when a rescuer had found a victim, and a medical agent is supposed to go to the place where the victim is found;
- Transition T2 “**Victim is found**” – represents the event when a rescuer agent has found a victim. It is connected to the T1, as the medical agent needs to start acting only when the victim is found by the rescuer;
- Transition T3 “**SOS**” – a rescuer has stuck in emergency;

The agents behaviour is determined by two element nets. The medical staff element net is depicted in Fig. 3 and the rescuer element net — in Fig. 6. In the real Search and Rescue operations there are usually more element nets and they are more detailed.

*Medical staff element net* represents what kind of actions should a medical agent perform while taking part in the operation. At first, the medical agent needs to get the medicine and learn about the operation. He will not be allowed to the next stage of the operation before he performs both of these actions. After doing that, he is supposed to wait until he receives the notification on the accident. Then he has to send his arrival time, and start making his way to the place where the accident had happened. The next two steps are to report the victim condition and to transport the victim to the infirmary. The medical agent also may contact coordinator at any time.

*Rescuers element net* is the model of part-taking for rescuers. Before entering the operation, each agent is required to do the following: get the equipment; obtain the information about other agents; and, get briefing about the operation. The equipment consists of three parts; and, the agent must equip them all. After entering the operation, the agent has to go to the exploration area. If a victim is found, the agent is supposed to send a photo, a description, and the accurate coordinates of the victim location. If something goes wrong, the agent can just send the location and the coordinator will handle it. Once the exploration is completed, he can receive the coordinates of the new area to explore.

#### IV. ARCHITECTURE

The way this system is designed relies on three basic components:

- NPNtool (Eclipse plugin) [7] for creating Petri Nets model and linking AL code to the transitions. The main purpose of this tool is to model a system net and element nets which will represent the model of the bot. The AL code will be linked to the transitions and then compiled to the executable file according to the model;
- Java-library consists of AL-compiler and AL-linker. AL-linker traces the system and element nets, collects all the code from the transitions, and eventually converts in to a text file that will be compiled by the AL-compiler. AL-compiler is created with the ANTLR[12] tool. AL-compiler gets an input text file and translates it to the executable artifact that actually represents the Telegram bot;
- Telegram Bot API library that consists of the code for requesting data via HTTP-requests from the Telegram Bot API server.

The overall technology chain is as follows. At first, a developer creates and verifies the NP-net model of a system via NPNtool. When the model is constructed, the developer inscribes AL code to the transitions according to the expected logic of the bot. Then the developer launches AL-linker which traverses the constructed NP-net collecting the textual representation of the transitions AL code into a text file. After that, AL-compiler reads the artifacts generated by AL-linker and generates a Telegram Bot code. The codegeneration of distributed systems from NP-nets models has been studied in [13]. Once a JAR file is compiled from that code, it could be executed. All the actions of the agents are displayed on the

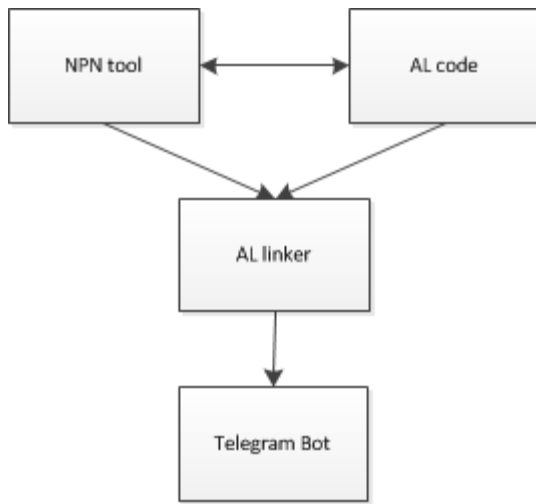


Fig. 2. The developed system workflow

bot host and could be processed at real-time (saved or directly answered) by the coordinator who ran the Bot.

Telegram bot consists of TBA library and several Java classes. Each Java class corresponds to an element net or a system net and stores a number of methods corresponding to the transitions with AL-code inscribed to them. These methods will use TBA to interchange the information. There is also a class that links all the element and system nets libraries together and proceeds the logic using event-based paradigm and asynchronous requests.

It shall be noticed that the compiled Telegram Bot is a server that communicates with the software clients — the rescue and medical staff software mobile clients. The bot is connected to the Telegram server via the webhook technology; namely, all the requests that agents send to the Telegram server via Telegram mobile applications are redirected to and served on the deployed bot server.

The fragment of code represents the method which corresponds to one of the Medical Staff element net transition:

```

public void taskReportVictimsCondition(String
mes, String chatId) throws
TelegramApiException{
    SendMessage message = new SendMessage();
    String[] tasks = {"Report about the
    victim's condition"};
    ReplyKeyboardMarkup replyKeyboardMarkup
    = makeKeyboard(tasks);
    message.setReplyMarkup(
    replyKeyboardMarkup);
    message.setText(mes);
    sendTo(message, chatId);
}
  
```

## V. ACTION LANGUAGE

AL compiler has been developed with ANTLR compiler which enables to define a grammar in a ANTLR grammar language and compile it to the Java classes which represent the lexer and parser of AL. The code generated by ANTLR

is able to build the syntax tree of AL code. To execute semantical actions while tracing through the nodes of this tree, the package visitor was created that contains classes for generating Java code from AL code.

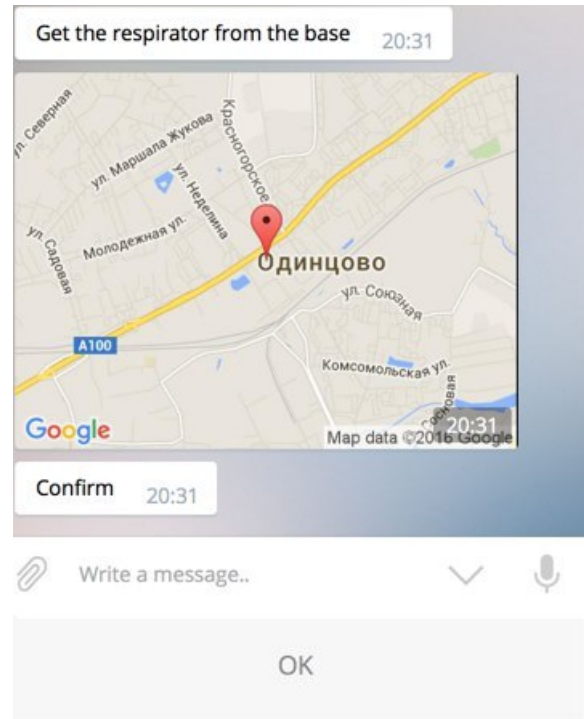


Fig. 3. An agent is confirming the task implementation

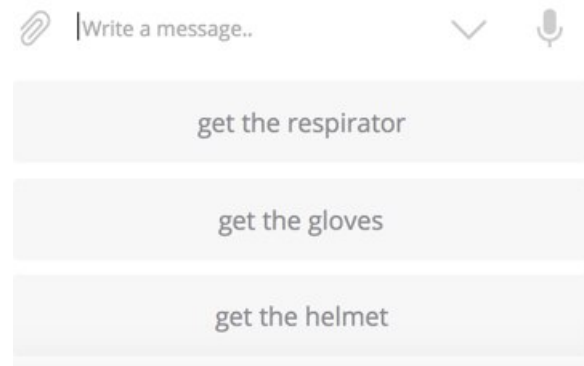


Fig. 4. The rescuer agent Telegram mobile client.

### A. AL grammar

- $\langle \text{initialization} \rangle ::= \langle \text{variable} \rangle = \langle \text{value} \rangle$  — it is possible to assign variables of the types  $\langle \text{file} \rangle$ ,  $\langle \text{float} \rangle$ ,  $\langle \text{string} \rangle$
- $\langle \text{SystemNet} \rangle ::= \langle \text{SP} \rangle : \langle \text{name} \rangle$  — the name of system net
- $\langle \text{ElementNet} \rangle ::= \langle \text{EP} \rangle : \langle \text{name} \rangle$  — the name of element net
- $\langle \text{file} \rangle ::= \text{file}(\langle \text{text} \rangle)$  — loads file from file-system

- `<sendMessage>::= sendMessage( <file>| <text>| <variable> )` — sends a message from a transition of an element net
- `<sendPhoto>::= sendPhoto(<file>| <variable> )` — send a Photo from a transition of an element net
- `<sendLocation>::= sendLocation(longitude : <variable>| <float>, latitude : <variable>| <float>)` — sends Location
- `<sendVideo>::= sendVideo(<file>| <variable> )` — sends Video
- `<sendAudio>::= sendAudio(<file>| <variable> )` — sends Audio
- `<transition element net>::= <name>= <text>`  
response: `( <sendAudio>| <sendVideo>| <sendLocation>| <sendPhoto>| <sendMessage> )*`  
— this is the structure of the code which should be inscribed on the distinct transition of an element net.
- `<connect>::= connect ( <name>.<transition> )` — links a transition from an element net to a transition of a system net.
- `<display>::= display()` — displays the object received on a transition of a system net
- `<save>::= save(<file>| <text>| <variable> )` — saves the object received on the transition of a system net
- `<transition system net>::= <name>= (receive (photo | video | audio | message | location) : (save | display)*)*`
- `<loop>::= forall <variable>in <variable botVariable>.add(<variable>)`

The AL example is actually based on our model which was provided in the motivating example. It illustrates what kind of code must be inscribed to the transition of the Medical staff element net (Fig. 7) and the coordinator system net (Fig. 8). We will not provide the code for Rescue element net because it follows the same pattern of coding as for the Medical staff element net.

## VI. APPLICATION OF THE TELEGRAM BOT CODEGENERATION TECHNOLOGY

In this section, we examine the application of the suggested technology to the motivating example provided in the section III. The main components of the system are modelled with system net and element nets. Then the codegenerator translates NP-nets into Telegram bots components of the target Telegram-based multi-agent system being constructed.

The bot server serves the received requests according to the NP-net system net behaviour and sends the answers to the agents. All the actions, except the actions described on the system net transitions, of the developed Search and Rescue operation are handled by the Bot automatically. However, it is possible to interact ad-hoc during the operation, i.e. if an agent sends any kind of request that was not described by AL, the coordinator will be notified and will be able to answer this request with the standard Telegram client interface. All

the phases that were described on the system net transitions require the direct interaction of the coordinator. The agent will not be allowed to proceed to the next stage of operation, unless he receives the answer from the coordinator.

As soon as we launch the compiled bot, all the rescuers and medicals that were loaded to the system will receive notifications from the Telegram bot. The concurrent transitions (e.g. Helmet, Respirator, Gloves) from the Rescuer element net allow that all the actions inscribed on them could be executed by agents in any order. An agent will not be allowed to the next stage unless he performed all of them. After performing an action, the agent must confirm that in the mobile client by pressing the OK button (Fig. 3). The button appears on the screen after every time the agent has actions-transitions to fire.

When an agents reaches the “Begin the operation” action, the bot moves to the awaiting state and notifies the coordinator, that the agent has reached the state and waits till the next instructions will be given. As soon as the coordinator fill the form and submit the answer, the agent will be allowed to move to the next state of his plan. That happens because the Begin the operation transition is synchronized with the T2 system net transition.

## VII. RELATED WORK AND FURTHER DIRECTIONS

The codegeneration from models to executable software artifacts has attracted attention when model driven development became industrial valuable approach [14]. The codegeneration from Petri net like models to executable software systems is studied for many formalisms and semi-formal industrial modelling languages like UML[15], [16] and SDL[17]. In [18], [19] the code generation tool for Input-Output Place-Transition Petri Nets was developed. In [20] the application of Sleptsov nets for modelling and implementation of hardware systems is studied. In [21] the technology to construct embedded access control systems from coloured Petri nets models is suggested. The approach to generate C++ code from SDL models is developed in [17]. The code generation from the UML state machines[15] and sequence [16] diagrams to executable code was studied. These are a lot of studies in the field, so we only cited a few.

The translation from NP-nets to coloured Petri nets was developed in [8]. The translation from NP-nets to PROMELA models to verify the correctness of LTL properties is studied in [22]. The automatic translation from NP-nets models to distributed systems components that preserve liveness, conditional liveness, and safety properties was studied in [13]. In the current work, we adopted the translation scheme developed in the latter work to obtain executable code from the structure of NP-nets models.

The further research concerns theoretical as well as practical aspects of the developed automatic codegeneration system. From the theoretical point of view, it is interesting to study preservation of different behavioural properties by the implemented translation and securing different behavioural consistencies of generated systems. As the underlying technologies are too large to conduct exhaustive formal verification, the both dynamic and static behavioural analyses techniques should be applied to study the correctness of the translation. From the practical point of view, there are lot of attractive features that



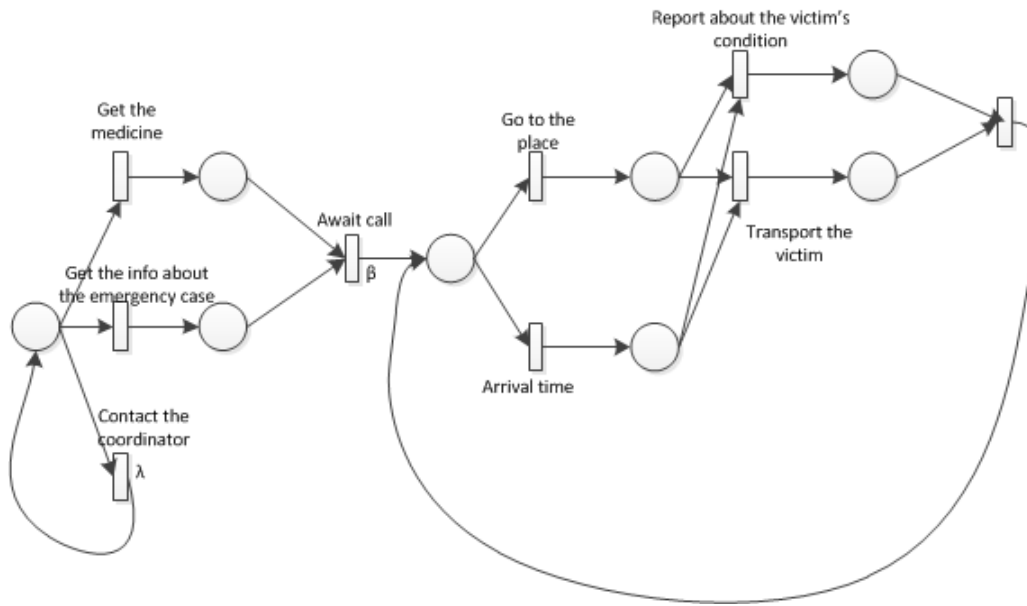


Fig. 5. The Medical Staff element net.

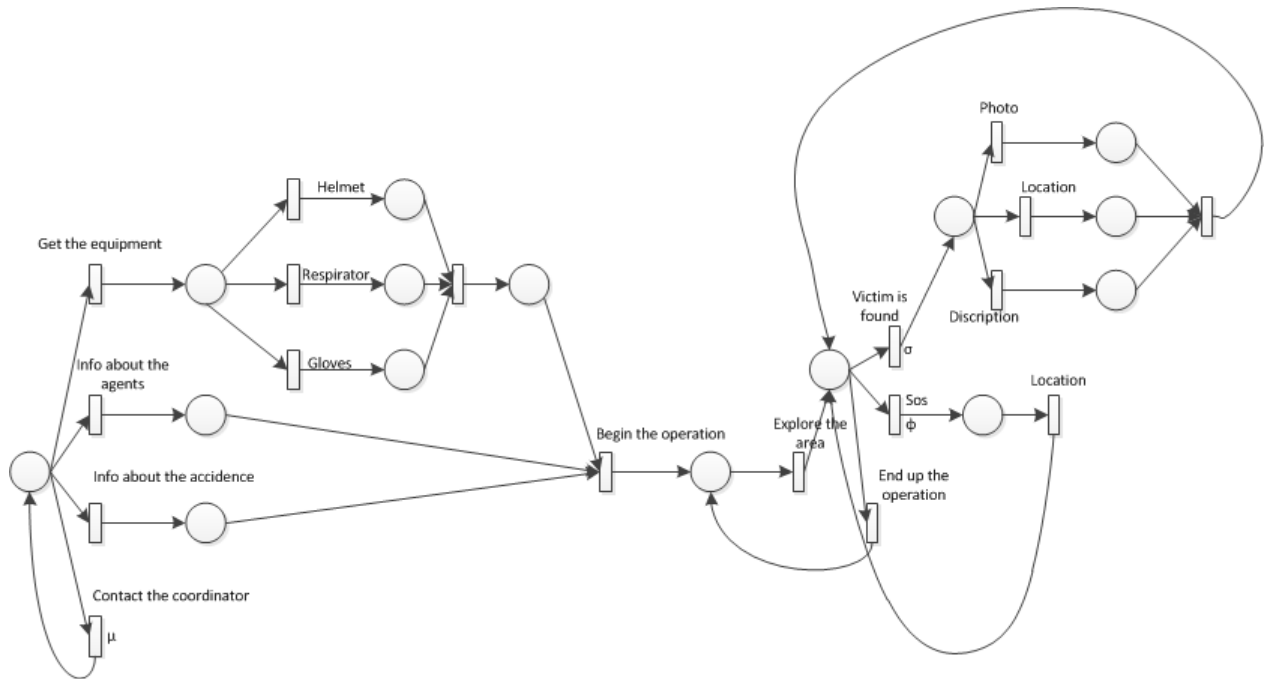


Fig. 6. The Rescuer element net.

are to be implemented. For example, it is not possible to change the deployed bots at runtime in the tool. However, such function could be of use for long term operations, when new actions should be integrated into an operating Telegram system without recompiling the whole system. The runtime deployment will be considered in the future research. Also, the scalability of generated Telegram systems and possible schemes of agents distribution in the system are the subjects of the further research.

## VIII. CONCLUSION

The developed technology enables developers to create Telegram Bots according to a visually clear model that could be verified and tested with help of the developed methods [22], [8], [9]. It allows to create distributed event-based Telegram Bots systems that operate on the Telegram platform and the AL language supports all the features provided by Telegram Bot API up to the moment.

The automatic code-generation reduces the risk of introducing defects on the implementation phase of software

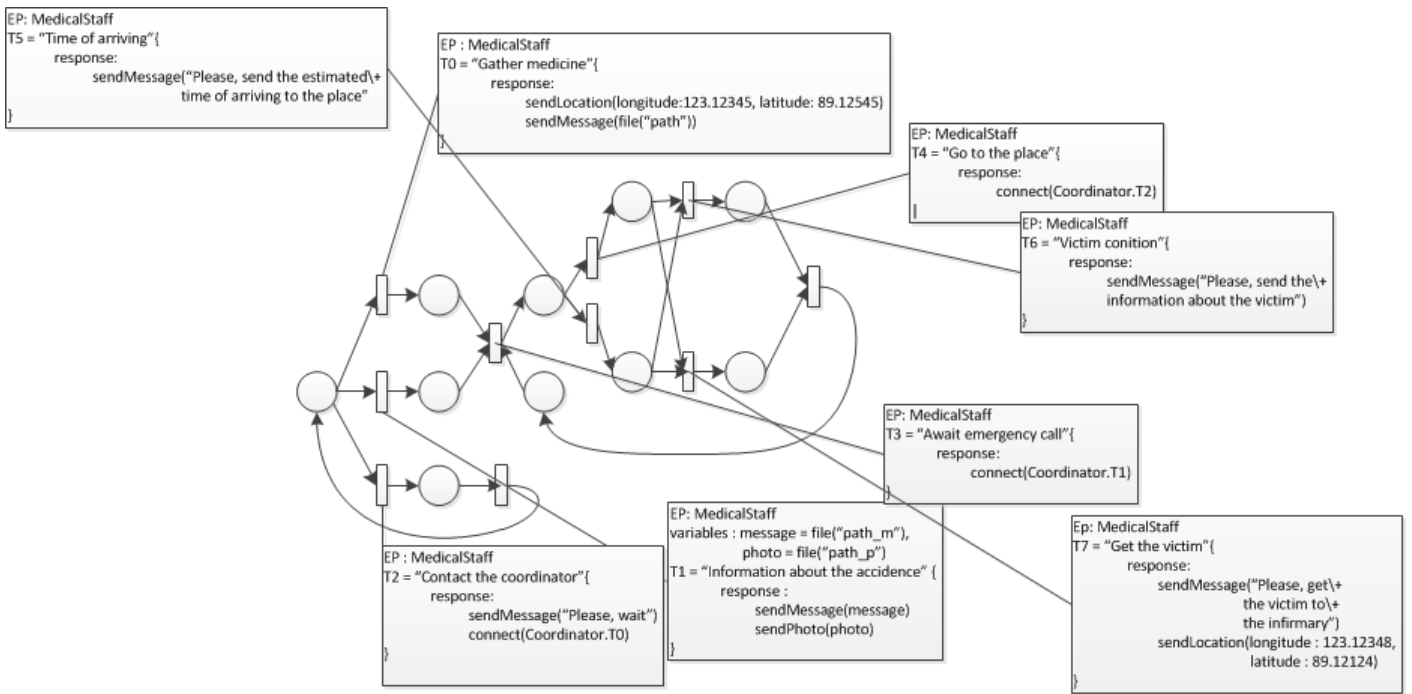


Fig. 7. The element net augmented with code

development process and improves the quality of the resultant code. It not only reduces the cost of software production, but also makes the quality of developed systems more predictable. The suggested technology is demonstrated with the example of a Search and Rescue system.

The authors would like to thank the anonymous referees for valuable and helpful comments.

#### ACKNOWLEDGMENT

This work is supported by the Basic Research Program at the National Research University Higher School of Economics and Russian Foundation for Basic Research, project No. 16-01-00546.

#### REFERENCES

- [1] (2016) Telegram Bot API online documentation. [Online]. Available: <https://core.telegram.org/bots/api>
- [2] L. Chang, X. He, J. Li, and S. M. Shatz, "Applying a nested Petri net modeling paradigm to coordination of sensor networks with mobile agents," in *Proc. of Workshop on Petri Nets and Distributed Systems*. Xian, China, 2008, pp. 132–145.
- [3] I. A. Lomazova, "Nested Petri nets - a formalism for specification and verification of multi-agent distributed systems," *Fundamenta Informaticae*, vol. 43, no. 1, pp. 195–214, 2000.
- [4] —, "Nested Petri nets: Multi-level and recursive systems," *Fundamenta Informaticae*, vol. 47, no. 3-4, pp. 283–293, Oct 2001.
- [5] —, "Nested Petri nets for adaptive process modeling," in *Pillars of Computer Science*, ser. Lecture Notes in Computer Science, A. Avron, N. Dershowitz, and A. Rabinovich, Eds. Springer Berlin Heidelberg, 2008, vol. 4800, pp. 460–474.
- [6] K. Hoffmann, H. Ehrig, and T. Mossakowski, "High-level nets with nets and rules as tokens," in *ICATPN*, 2005, pp. 268–288.
- [7] D. Frumin and L. Dworzanski, "NPNtool: Modelling and analysis toolset for nested Petri nets," in *Proceedings of the 7th Spring/Summer Young Researchers Colloquium on Software Engineering*, 2013, pp. 9–14.
- [8] L. Dworzanski and I. Lomazova, "CPN tools-assisted simulation and verification of nested Petri nets," *Automatic Control and Computer Sciences*, vol. 47, no. 7, pp. 393–402, 2013. [Online]. Available: <http://dx.doi.org/10.3103/S0146411613070201>
- [9] —, "On compositionality of boundedness and liveness for nested Petri nets," *Fundamenta Informaticae*, vol. 120, no. 3-4, pp. 275–293, 2012.
- [10] (2016) The ministry of the russian federation for civil defence, emergencies and elimination of consequences of natural disasters. emergency cases registered in russia. [Online]. Available: <http://25.mchs.gov.ru/document/2644168>
- [11] (2013) United states coast guard search and rescue summary statistics. [Online]. Available: <https://www.uscg.mil/hq/cg5/cg534/SARfactsInfo/SAR%20Sum%20Stats%2064-13.pdf>
- [12] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.
- [13] L. Dworzanski and I. Lomazova, "Automatic construction of distributed component system from nested Petri nets (in Rus)," 2016, in print, Programmirovaniye, ISSN: 0361-7688.
- [14] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, p. 19, 2003.
- [15] A. Knapp and S. Merz, "Model checking and code generation for uml state machines and collaborations," *Proc. 5th Wsh. Tools for System Design and Verification*, pp. 59–64, 2002.
- [16] D. Kundu, D. Samanta, and R. Mall, "Automatic code generation from unified modelling language sequence diagrams," *Software, IET*, vol. 7, no. 1, pp. 12–28, 2013.
- [17] P. Morozkin, I. Lavrovskaya, V. Olenov, and K. Nedovodeev, "Integration of sdl models into a systemic project for network simulation," in *SDL 2013: Model-Driven Dependability Engineering: 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*. Springer Berlin Heidelberg, 2013, pp. 275–290.
- [18] L. Gomes, J. P. Barros, A. Costa, and R. Nunes, "The input-output place-transition Petri net class and associated tools," in *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 1. IEEE, 2007, pp. 509–514.
- [19] R. Campos-Rebelo, F. Pereira, F. Moutinho, and L. Gomes, "From IOPT Petri nets to C: An automatic code generator tool," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*. IEEE, 2011, pp. 390–395.

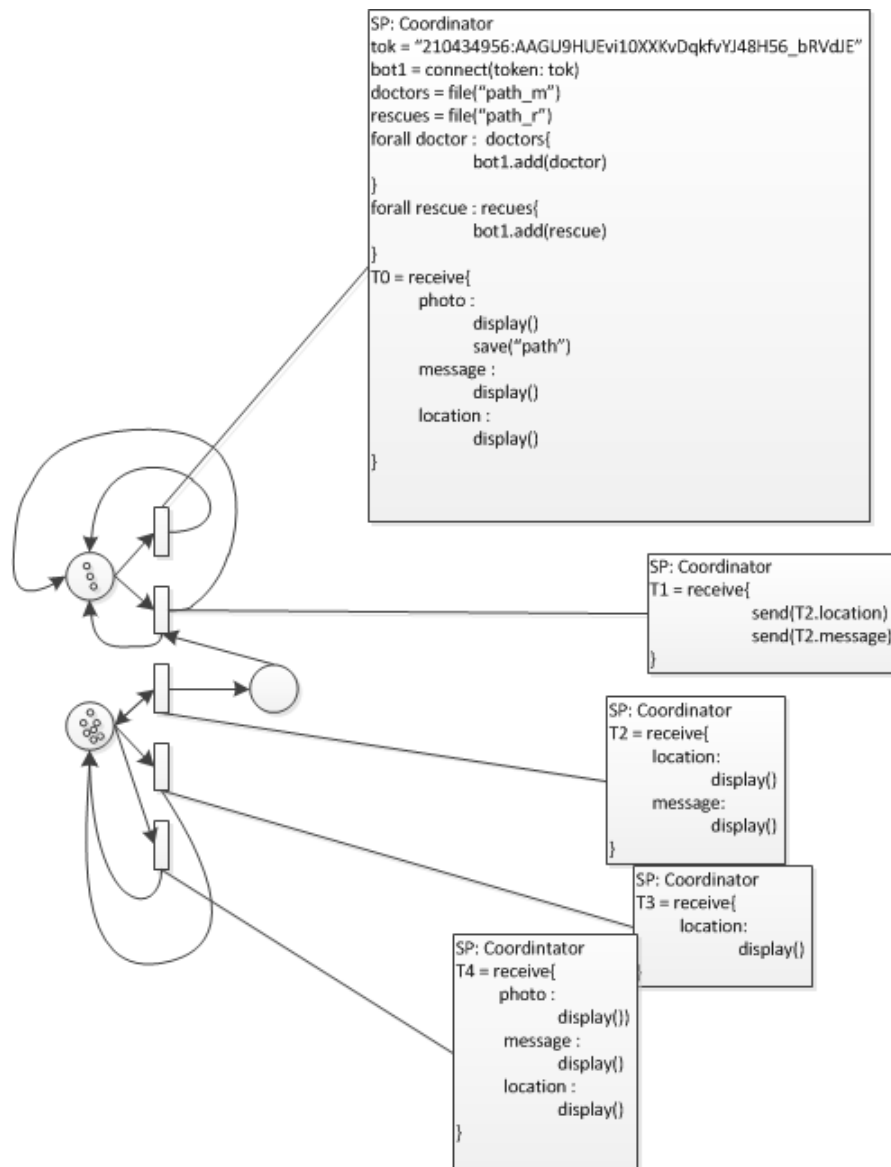


Fig. 8. The system net augmented with code

- [20] D. Zaitsev and J. Jürjens, "Programming in the sleptsov net language for systems control," *Advances in Mechanical Engineering*, vol. 8, no. 4, p. 1687814016640159, 2016.
- [21] K. H. Mortensen, "Automatic code generation method based on coloured petri net models applied on an access control system," in *Application and Theory of Petri Nets 2000*. Springer, 2000, pp. 367–386.
- [22] M. L. F. Venero and F. S. C. da Silva, "Model checking multi-level and recursive nets," *Software & Systems Modeling*, pp. 1–28, 2016.

# Mining Hierarchical UML Sequence Diagrams from Event Logs of SOA systems while Balancing between Abstracted and Detailed Models

Ksenia V. Davydova  
National Research University  
Higher School of Economics,  
PAIS Lab. at the Faculty of Computer Science,  
20 Myasnitskaya st.  
Moscow, 101000, Russia  
Email: kvdavydova@edu.hse.ru

Sergey A. Shershakov  
National Research University  
Higher School of Economics,  
PAIS Lab. at the Faculty of Computer Science,  
20 Myasnitskaya st.  
Moscow, 101000, Russia  
Email: sshershakov@hse.ru

**Abstract**—In this paper we consider an approach to reverse engineering of UML sequence diagrams from execution traces of SOA information systems represented as event logs. UML sequence diagrams are suitable for representing interactions in heterogeneous component systems; in particular, they include increasingly popular SOA-based information systems. In this paper we consider a new approach to inferring UML sequence diagrams from execution traces. They are logged by almost all modern information systems to so-called event logs. In contrast with conventional reverse engineering techniques which require source code for their work, our approach deals with event logs only. The approach consists of several parts of building UML sequence diagrams according to different perspectives and having different structures. They include mapping log attributes to diagram elements with an ability to set a level of abstraction and build hierarchical diagrams. We evaluate the approach in a software prototype implemented as a Microsoft Visio Add-In. The Add-In builds a UML sequence diagram from a given event log according to a set of customizable settings.

**Index Terms**—Event log, UML sequence diagram, reverse engineering.

## I. INTRODUCTION

Nowadays there are a lot of information systems. They are developed by people which are error-prone. Systems also can have a difficult to understand structure. Thus, models are necessary to understand systems and find errors. When there is no complete model of a system, reverse engineering techniques can be applied to extract necessary information from the system and build an appropriate model. There are a number of tools for this purpose, they analyze source code of the system and build a model.

There are some types of models which are useful to analyze in software engineering. For example, state machines are able to model a large number of software problems. However, they have a weakness in describing an abstract model of computation. Another example of a software model is Petri nets which can describe processes with concurrent execution. Furthermore, there is a number of models described by a standard of Unified Modeling Language (UML) for visualizing design of information systems. UML 2.4.1 [1] has two groups of diagrams, structural and behavioral ones. In particular, such kind of UML diagrams as *state class diagrams*, *statecharts* and *sequence diagrams* are widely applied to reverse engineering domain.

Almost every information system has an ability to write results of its execution to event logs. We propose approaches to mine UML sequence diagrams (UML SD) from these logs. Event logs

of information systems with a service-oriented architecture (SOA) are considered and UML SD are applied to modeling interaction between SOA information system components.

In contrast to existing reverse engineering tools which use source code, we work with *system execution traces* in the form of event logs. A technique that allows analysis of business processes based on event logs is called process mining [2]. It uses specialized algorithms for extracting knowledge from event logs recorded by an information system. Moreover, process mining helps to check the conformance of a derived model with its earlier specification. Using execution traces works even if there is no access to the source code of an information systems. Also, not all versions of code are normally stored. Moreover, large information systems tend to be distributed. Different components of a system are often implemented using different programming languages. Such a problem is solved by considering event logs instead of source code.

### A. Motivating example

There is an event log written by a SOA-based banking information system (Table 1). We are interested in building a model in the form of a UML sequence diagram reflecting processes in the system. We have only some of the runs of the process, so one of the problems is to build an as feasible model as possible. The log contains a number of execution traces. Each trace consists of a sequence of events ordered by Timestamp attribute. Columns represent attributes of the log and rows represent its events. System executions are maintained by different components of the system. They are grouped in attributes such as *Domain*, *Service/Process* and *Operation*. *Domains* group *Services* and *Processes*, and the latter consist of *Operations* [3].

Interaction between program system components can be represented at different abstraction levels. For example, by mapping some log attributes onto structural elements of UML SDs, such as lifelines and messages, one can get a UML SD diagram such as on Figure 1. Specific values of these attributes appear with head names such as “Domain::Service/Process”. Similarly, values of *Operation* and *Payload* attributes which are mapped onto messages parameters appear with message arrows. Timestamp attribute sets an order of calls (time goes from the top to the bottom of a diagram).

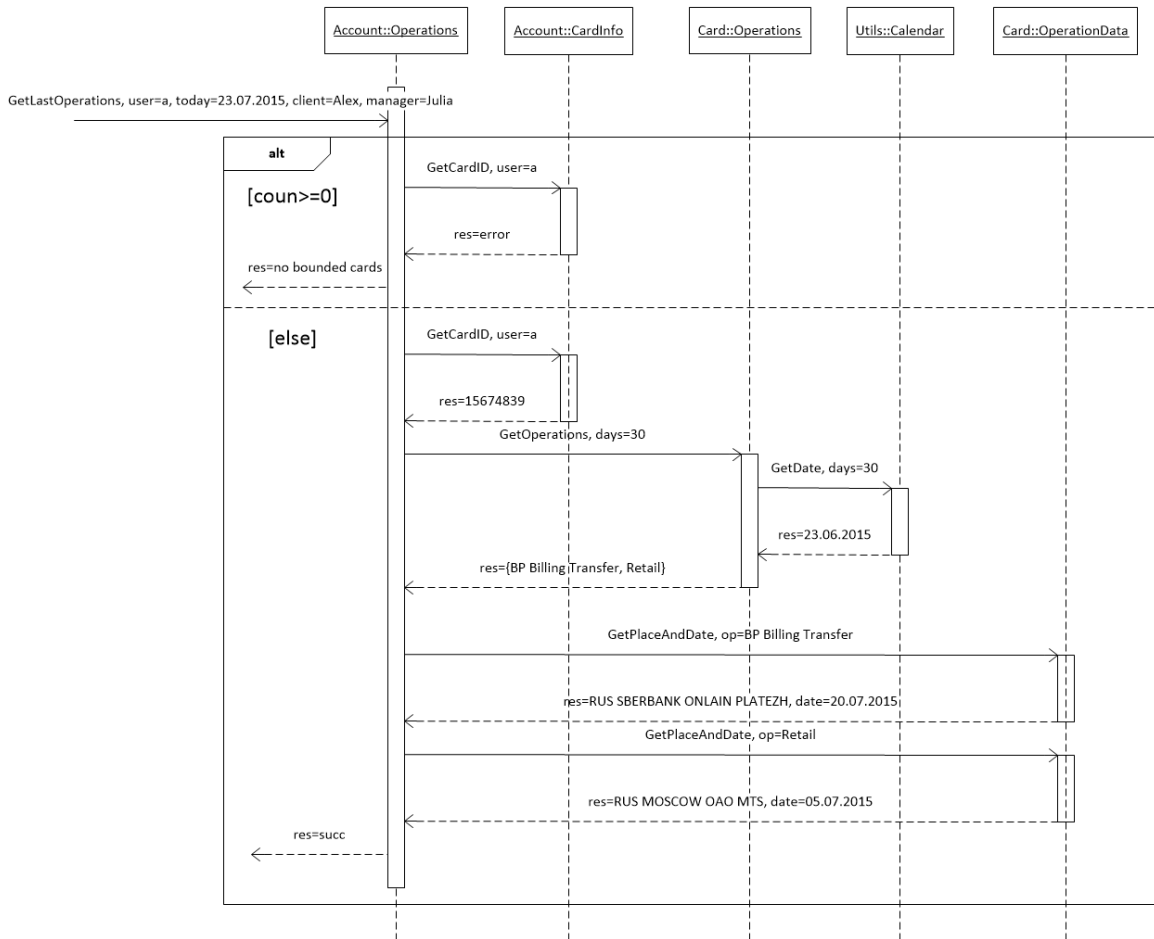


Fig. 1. Mapping log attributes onto UML sequence diagram components

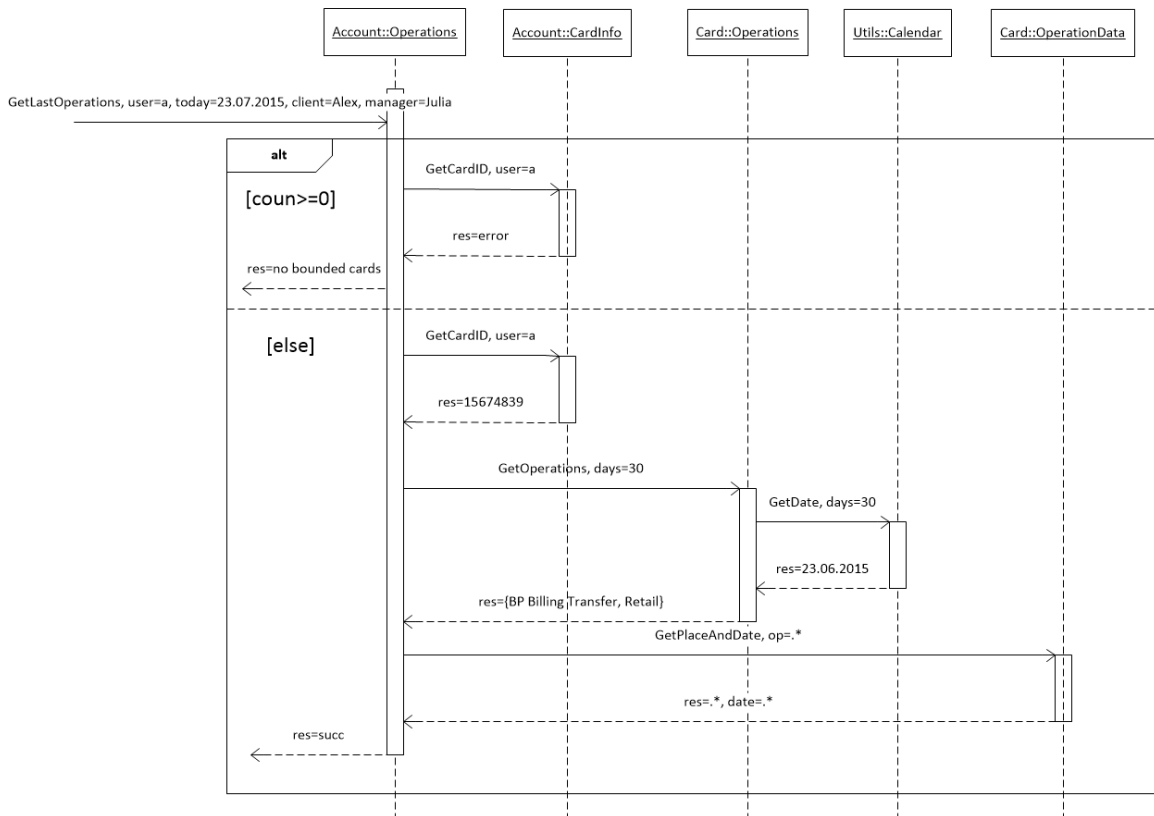


Fig. 2. Merge of diagram components based on a regular expression

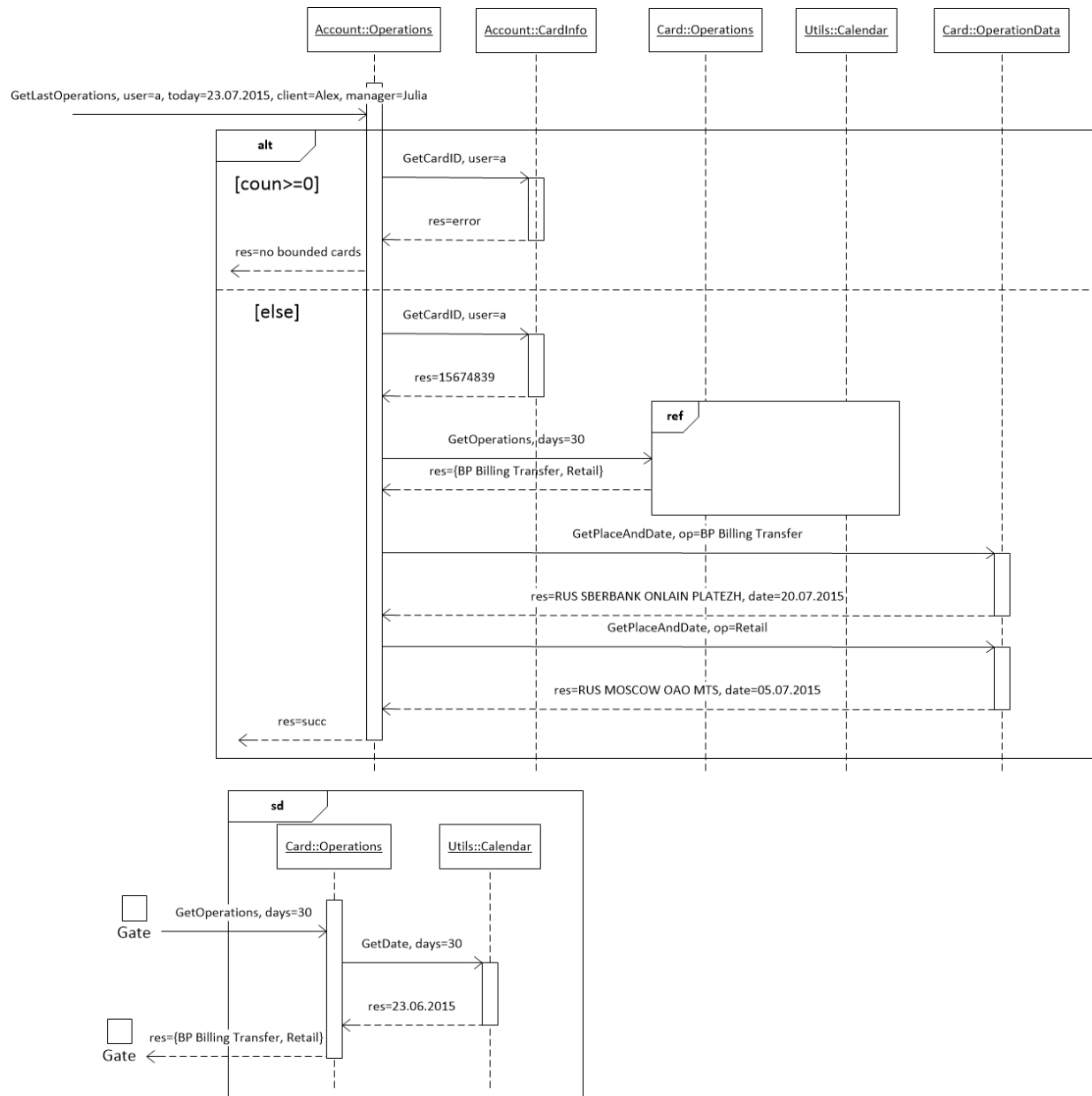


Fig. 3. Hierarchical UML sequence diagram using nested fragments

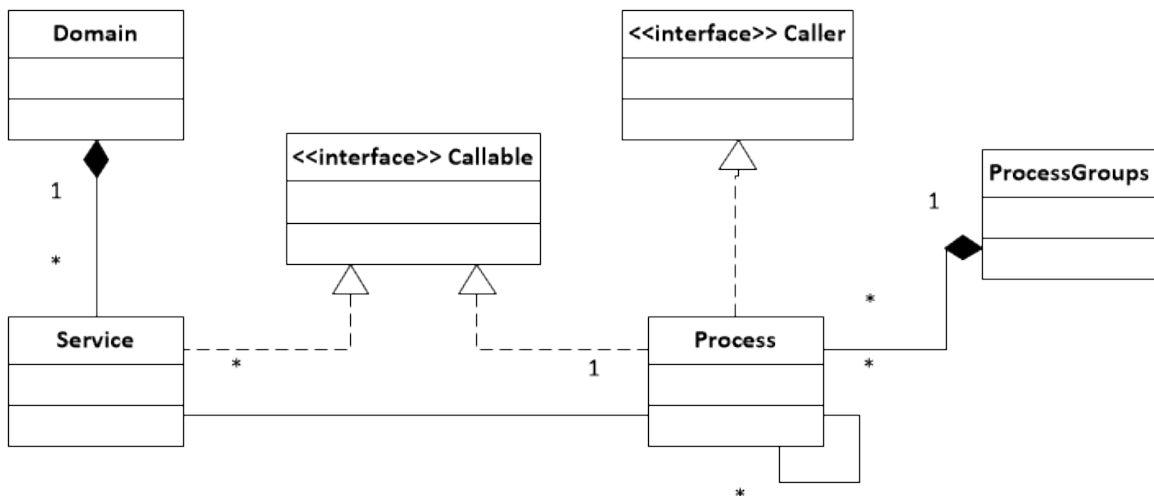


Fig. 4. Meta-model of a SOA system

TABLE I. Log fragment L1. Banking SOA-system

CaseID	Domain	Service/Process	Operation	Action	Payload	Timestamp
23	Account	Operations	GetLastOperations	REQ	user=a, today=23.07.2015, client=Alex, manager=Julia	17:32:15 135
23	Account	CardInfo	GetCardID	REQ	user=a	17:32:15 250
23	Account	CardInfo	GetCardID	RES	res=15674839	17:32:15 297
23	Card	Operations	GetOperations	REQ	days=30	17:32:15 378
23	Utils	Calendar	GetDate	REQ	days=30	17:32:15 409
23	Utils	Calendar	GetDate	RES	res=23.06.2015	17:32:15 478
23	Card	Operations	GetOperations	RES	res={BP Billing Transfer, Retail}	17:32:15 513
23	Card	OperationData	GetPlaceAndDate	REQ	op=BP Billing Transfer	17:32:15 589
23	Card	OperationData	GetPlaceAndDate	RES	res=RUS SBERBANK ONLAIN PLATEZH, date=20.07.2015	17:32:15 601
23	Card	OperationData	GetPlaceAndDate	REQ	op=Retail	17:32:15 638
23	Card	OperationData	GetPlaceAndDate	RES	res=RUS MOSCOW OAO MTS, date=05.07.2015	17:32:15 735
23	Account	Operations	GetLastOperations	RES	res=succ	17:32:15 822
25	Account	Operations	GetLastOperations	REQ	user=a, today=23.07.2015, client=Alex, manager=Julia	17:40:18 345
25	Account	CardInfo	GetCardID	REQ	user=a	17:40:18 408
25	Account	CardInfo	GetCardID	RES	res=error	17:40:18 489
25	Account	Operations	GetLastOperations	RES	res=no bounded cards	17:40:18 523

It can also be useful to merge some messages or lifelines in order to reduce the size of a diagram and avoid “spaghetti-like” models. A regular expression suits it and the example of their usage is depicted on Figure 2.

Some interaction sometimes can be useful to picture on one diagram and other interactions on nested diagrams. Those diagrams use an interaction fragment labeled *ref*. An example of that hierarchical diagram is on Figure 3.

It would be good to have a tool which can do mapping of event log attributes on UML sequence diagram elements with ability to set an abstraction level for seeing different perspectives of the system execution. An approach approved in VTM4Visio framework is applied, which allows building these diagrams.

### B. Related work

Reverse engineering of UML sequence diagrams is not a new problem. There are a number of works, such as [4], [5], [6], [7], applied static approaches (getting models from source code without execution) for solving this problem. Moreover, there is a number of CASE tools for reverse engineering of UML sequence diagrams and other types of UML diagrams. However, most of them use static program analysis without execution of a program. Static program analysis usually uses source code or object code (a result of source code compilation). Some of these tools analyze source code, some of these tools analyze both source code and object code. However, event logs are execution traces of source code. Thus, we do not need access to source code.

The most popular CASE tools are Sparx Systems’ Enterprise Architect [8], IBM Rational Software Architect [9], Visual Paradigm [10], Altova UModel [11], MagicDraw [12], StarUML [13], ArgoUML [14]. There are both tools for end-to-end design and simple UML editors. The former include Sparx Systems’ Enterprise Architect, IBM Rational Software Architect, Visual Paradigm, Altova UModel and MagicDraw, the latter include StarUML and ArgoUML. Beside that, the main aim of these tools is to get models from source code. Table II [15] contains CASE tools and program languages, for which models can be built. As we can see, none of these tools is able to infer models from the most popular languages used for developing SOA information systems. Moreover, a SOA architecture can be developed with various programming languages. For example, some modules

TABLE II. Programming languages of reverse engineering tools

Tools	Programming languages						
	PHP	C++	Java	Ruby	Python	VB	C#
Sparx Systems’ Enterprise Architect	+	+	+	-	+	+	+
IBM Rational Software Architect	-	+	-	-	-	+	+
Visual Paradigm	+	+	+	+	+	-	+
Altova UModel	-	-	+	-	-	+	+
MagicDraw	-	+	+	-	-	-	+
StarUML	-	+	+	-	-	-	+
ArgoUML	-	+	+	-	-	-	+

can be written in C#, others can be developed in Java, they can interact with LAMP service, so a single CASE tool cannot produce models for that system. Mining diagrams from event logs solves this problem.

There are some works, such as [16], [17], [18], [19], where approaches are applied for building UML sequence diagrams from program system execution traces (dynamic approaches). One of related works [16] analyzes one trace using a meta-model of the trace and a UML SD. The trace includes information not only about invocation of methods but also about loops and conditions, which makes easier recognition of fragments such as iteration, alternatives and option. However, program systems logging does not usually include this information, so it is necessary to change source code to apply this approach. In opposite to this approach, our approach recognizes fragments as conditions based on traces’ difference.

There is a dynamic approach to build a UML sequence diagram based on multiple execution traces in [18]. The authors apply an approach to build a Labeled Transition System (LTS) from a trace and an algorithm to merge some LTSs into one. After that the LTS is transformed into a UML sequence diagram. In opposite to this approach, we propose not to use other data structures to represent traces and merge them. We propose to map traces onto a UML sequence diagram directly without intermediate models, which is more efficient.

In [19] the authors pay more attention to analysis of derived models. They describe an approach briefly, without details. They mention that diagrams of one trace are merged into one UML sequence diagram. However, there is no mathematically strict definition of a trace or a UML sequence diagram and it is not



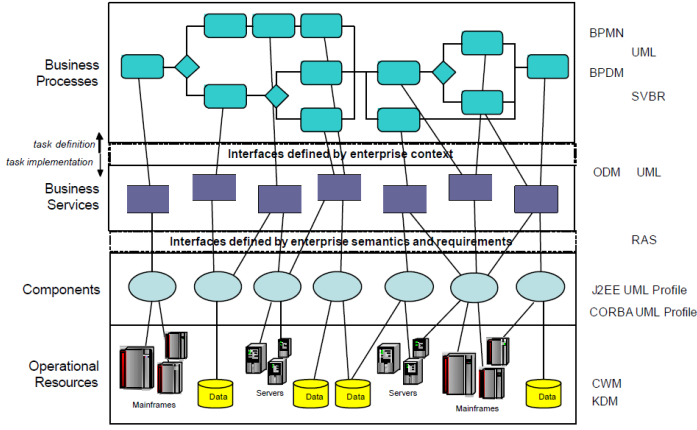


Fig. 5. Service-Oriented Architecture structure

clear how they merge several diagrams.

The rest of the current paper is organized as follows. Section II gives definitions. Section III introduces our approach to mining UML sequence diagrams. Section IV discusses results of some experiments on deriving models with the help of the developed tool. Section V concludes the paper and gives directions for further research.

## II. PRELIMINARIES

**Definition 1. (Event log)** Let  $E$  be a set of events. An event is a tuple  $e = (a_1, a_2, \dots, a_n)$ , where  $n$  is a number of attributes.  $\sigma = \langle e_1, e_2, \dots, e_k \rangle$  is an event trace (i.e. an ordered set of events which normally belongs to one case).  $Log = \mathcal{P}(E)$  is an event log which is a multi-set of traces.

In the paper we consider primarily event logs written by SOA information systems. Logs have a structure according to a SOA systems standard. A meta-model of such systems is depicted on Figure 4. The model complies with a Service Oriented Architecture standard (Figure 5) proposed by Object Management Group [20].

We introduce a formal definition of a UML sequence diagram as follows.

**Definition 2. (UML Sequence Diagram)** A UML sequence diagram is a tuple  $U_{SD} = (L, A, M, T, P, Ref, \delta)$ , where:

- $L$  is a set of lifelines, they represent objects whose interaction is shown on the diagram.
- $A$  is a set of activations (emit and take messages) mapped onto lifelines.  $A \subseteq (L \times T \times T)$
- $T$  is time, it goes from the top of the diagram to the bottom.  $\forall t \in T, \delta(t) = y$ , where  $y \in \mathbb{Z}$
- $M$  is a set of messages (call and return) with its parameters and is ordered by time.  $M \subseteq ((A \cup Ref) \times T \times P \times (A \cup Ref))$   
 $m \in M : m = (a_1, t, p, a_2)$ , where  $a_1 \in A \cup Ref, t \in T, p \in P, a_2 \in A \cup Ref$   
 $a_1 = (l_1, t_{11}, t_{12}), a_2 = (l_2, t_{21}, t_{22}) : t_{11} < t_{21}, t_{11} < t_{12}, t_{21} < t_{22}$
- $P$  is a set of parameters of messages.
- $Ref$  is a set of *ref* fragments which group lifelines and hide them interaction. The interaction is shown on another diagram.
- $\delta : U_{SD} = (U_{SD1}, U_{HSD} | L' \subseteq L, L_1 \subseteq L, A' \subseteq A, A_1 \subseteq A, A_1 \cap A' = \emptyset$

$M' \subseteq M, M_1 \subseteq M, M_1 \cap M' = \{m = (a_1, t, p, a_2) | a_1 \in A_1, a_2 \in A'\}$   
 $P' \subseteq P, P_1 \subseteq P, P_1 \cap P' = \{p | m = (a_1, t, p, a_2), a_1 \in A_1, a_2 \in A', p \in P_1, p \in P'\}$   
 $Ref' \subseteq Ref, Ref_1 \subseteq Ref, Ref_1 \cap Ref' = \emptyset$ ,  
where :  
 $U_{SD} = (L, A, M, T, P, Ref)$  – a detailed diagram.  
 $U_{SD1} = (L_1, A_1, M_1, T, P_1, Ref_1)$  – a diagram with *ref* fragment.  
 $U_{HSD} = (L', A', M', T, P', Ref')$  – a nested diagram.

## III. APPROACHE TO BALANCE BETWEEN ABSTRACTION AND DETALISATION

We propose an approach to mining UML sequence diagrams from an event log with a various degree of detalization. The approach consists of three steps derived one from another. It is necessary to map attributes of the log onto elements of a diagram prior to beginning a mining procedure. Some mapping functions are therefore needed. First, it is necessary to define which interaction of SOA components (*Services, Processes, Domains* etc.) must be depicted on the diagram. Function  $\alpha$  (1) maps events of the log with their attributes onto lifelines of diagrams. It allows to choose attributes to be represented on the diagram as lifelines.

$$E = (e_1, e_2, \dots, e_k), k - a \text{ number of events} \quad (1)$$

$$\alpha : U(E) \rightarrow L$$

### A. Mapping log attributes onto UML sequence diagram components

The first step allows getting diagrams with different abstraction levels by choosing log attributes for mapping onto lifelines and attributes for mapping onto parameters. To map attributes onto lifelines function  $\alpha$  is used. Values of attributes *Domain* and *Service* are mapped onto composite lifeline objects with head names such as “Domain::Service/Process” on Figure 1. Also, function  $\gamma$  (2) is introduced for mapping attributes onto message parameters. *Operation* and *Payload* attributes are mapped onto messages parameters on Figure 1 such as “Operation, Payload”.

$$\gamma : U(E) \rightarrow P \quad (2)$$

The diagram depicted on Figure 1 demonstrates interaction of services. The model represents one of the possible configurations of abstraction for the event log in table I. For example, another possible configuration includes *Service/Process* and *Operation* attributes as diagram objects. Choosing such attributes allows to infer diagrams with different abstraction levels.

### B. Merge of diagram components

On Figure 1 we see that the last two invocations of *GetPlaceAndDate* function are almost equal except for operation parameters. The second step of our approach performs merging of some parts of a diagram. We propose to merge similar parts by using regular expressions. A regular expression contains a common part of a number of merged parts. The approach allows to reduce the size of a model by merging similar parts. It increases generalization of the model.

The approach involves a Cartesian product of a log to itself with filtering. Function  $\beta$  (3) is used to map a filtered Cartesian

product of a log to itself on the set  $\{1, 0\}$  so that the element of the product will be a pair “event” - “event from a set of next events”. If the pair satisfies a regular expression then it is marked as 1, otherwise as 0. We introduce  $\eta$  (4) to compare elements of the product.  $\eta$  considers events as equal to each other if their corresponding attributes are equal. In this case attributes are equal if they can be matched as a single regular expression. Functions  $\alpha$  and  $\gamma$  are used in this approach for mapping event attributes onto UML sequence diagram elements. There is also introduced function  $\xi$  (5) which determines a family of messages which are satisfied with pair event attributes. A message can be just a value of attributes or a regular expression applicable to single event attributes.

$$\beta : E \times E \rightarrow \{0, 1\} \quad (3)$$

$$\begin{aligned} e_i &= (a_{1,1}, a_{1,2}, \dots, a_{1,n}) - \text{an event with } n \text{ attributes} \\ &\quad (e_1, e_2) \in E \times E \\ \tilde{e}_1 &= (a_{1,1}, a_{1,3}, \dots, a_{1,p}) - \text{an event with } p \\ &\quad \text{sample attributes, } p < n \\ \tilde{e}_2 &= (a_{2,1}, a_{2,3}, \dots, a_{2,p}) - \text{an event with } p \\ &\quad \text{sample attributes, } p < n \\ \eta : \tilde{e}_1 = \tilde{e}_2 &\Rightarrow \\ a_{1,1} = a_{2,1} \& a_{1,3} = a_{2,3} \& \dots \& a_{1,p} = a_{2,p} \\ \forall m \in M \exists \tilde{e} \in E \times E : \xi(\tilde{e}) = m \& \beta(\tilde{e}) = 1, \\ M &- \text{set of messages} \end{aligned} \quad (4)$$

$$(5)$$

If one looks at an example introduced above on Figure 2, the diagram is obtained through applying this function and regular expressions. It is noticeable that two invocations of operation *GetPlaceAndDate* are merged in one invocation with regular expressions in message parameters. Regular expression “.” means that any sequence of symbols can be inserted instead of this expression. It is also possible to merge lifelines by using regular expressions. It can be useful if class *A* is invoked by only class *B*; so, these classes can be merged into one lifeline.

### C. Mining a hierarchical UML sequence diagram using nested fragments

One of the ways to represent a complex model is creating a hierarchical model. The UML standard [1] allows us to divide a complex diagram into more abstract and detailed models interacting through *gates*.

In order to define a hierarchy in a UML sequence diagram we introduce a definition of a selection criterion as follows. The definition of a hierarchical UML sequence diagram is given in Definition 2.

**Definition 3. (Selection criterion)** Let  $k$  be a number of hierarchical levels and  $RE$  be a regular expression defined in [21] with an added symbol “.” as an any symbol designation. Then,  $c = \langle c_i | c_i \in RE \rangle$ ,  $c_i$  is a selection criterion of events for  $i$ -hierarchical level.  $c = c_1 \cup c_2 \cup \dots \cup c_k$  and  $c_1 \cap c_2 \cap \dots \cap c_k = \emptyset$ . The regular expressions defined in [21] as selection criteria are boolean expressions because their abstract syntax includes Boolean operations.

The components of SOA systems described by a meta-model depicted on Figure 4 have hierarchical relationship with each other. According to the SOA model there is a hierarchy in

event log because processes invoke different subprocesses or services.

It is also possible to distinguish some technical sublevels from main level by applying regular expressions. We propose a previously defined step with regular expressions to group elements.

Each hierarchical level is able to be encapsulated into another level on a UML sequence diagram. We propose to use nested fragments labeled as *ref* which are defined in [1]. It allows combining high-level and detailed views of diagrams at the same time.

For applying the approach a number of hierarchical levels and selection criteria, which are defined in Definition 3, need to be specified. Function  $\beta$  defines whether two events can be grouped into a single sublevel. If events match a selection criterion then they are moved to a nested diagram. For this case values of some attributes must be equal or match a single regular expression. Function  $\delta$  (Definition 2) maps some part of a UML sequence diagram considered as nested on a separate UML SD. The mapping uses *interaction use* which is shown as a *combined fragment* with operator *ref* [1]. This fragment hides some details of a high-level diagram moved to a nested diagram while the referred diagram allows seeing details.

On Figure 3, a hierarchical UML sequence diagram for event log L1 is depicted there. There is some elements’ interaction on the high-level diagram and some interaction is abstracted as *ref* fragment and depicted on the nested one. A selection criterion used for building the diagrams is “*Operation=GetDate*” which defines a part to be abstracted.

## IV. EVALUATION

This section discusses our evaluation of the approach presented in this paper.

### A. VTM4Visio Framework

Microsoft Visio is a professional drawing tool for making business charts and diagrams. It also supports some of UML diagrams. Besides, Visio has reverse engineering of databases, but it does not support UML reverse engineering. One of flexible features is that it can be expanded by add-ins. It is possible to use Visio SDK [22] for having access to a Visio object model. Thus, it is a good solution to implement our tool for visualizing results (UML sequence diagrams) of our mining algorithm.

VTM4Visio is an extensible framework aimed at process mining purposing. It is implemented as an add-in for Microsoft Visio 2010. Our tool is implemented as a plug-in which is supported by one of the VTM4Visio components called Plugin Manager.

This framework was chosen because it provides useful instruments for accessing Microsoft Visio object models. It also has a convenient GUI.

### B. Log pre-processing

It is necessary to have an event log in a definite format to apply our algorithm. A lot of information systems write logs in their own format. Our algorithm requires the event log must contain attributes which can be used as a case id, timestamp and activity attributes. It is necessary to format and validate the event log before applying the algorithm.

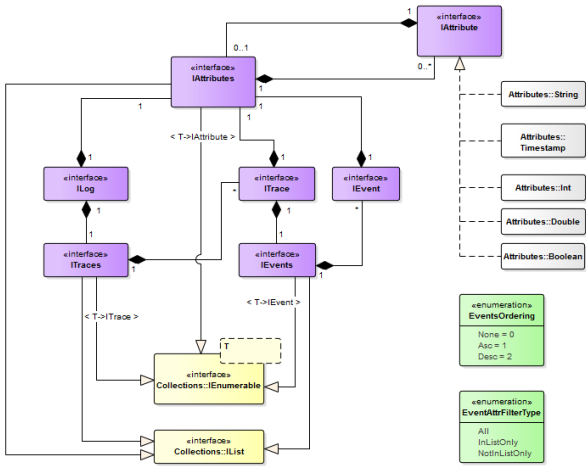


Fig. 6. Class diagram of Event log object model library

### C. Log library

Our algorithm requires an event log for mining a UML SD to be in some definite format. That is why it is necessary to have a library for work with event logs. We made the library and called it “Event Log Object Model Library” whose UML class diagram is depicted on Figure 6. The structure of our library is inspired by XES format [23]. It is not based on it but main components are taken from XES standard. We introduce special types such as *EvntsOrdering* and *EventAttrFilterType* for CSV and RDBMS-based event logs [24] because XML-based XES format is excessive. The library is written in C#. It is extensible, which allows working with different event log formats.

### D. Prototype implementation

Our prototype was written in C# programming language as a plug-in for VTM4Visio framework. The prototype allows to configuring parameters for our approaches as CaseID, Timestamp and Activity, names of lifelines and messages’ parameters, a regular expression through some GUI forms (Figures 7 and 8). The configuration for reading of event logs from a file is implemented as shown on Figure 7. The configuration of the diagram is implemented as shown on Figure 8. This GUI form allows setting different perspectives and a regular expression for merging diagram elements and, hence, specifying hierarchy.

The processing result of the event log in Table I is depicted on Figures 1, 2, 3.

## V. CONCLUSION

This paper proposes a method of reverse engineering of UML sequence diagrams from event logs of SOA information systems. It contains three approaches to balance high-level diagrams and low-level ones.

Our method is a dynamic analysis of software because it uses only event logs. This is an advantage since source code is not always available. Also, our approaches do not use intermediate models of an event log representation. The proposed method 1) maps log attributes onto diagram components, 2) merges diagram elements based on regular expressions and 3) builds hierarchical UML diagrams using a *ref* fragment.

Work with event logs of real-life SOA information systems shows that it is necessary to mine diagrams not only from single-threaded event logs but also from multi-threaded ones. Thus, it is

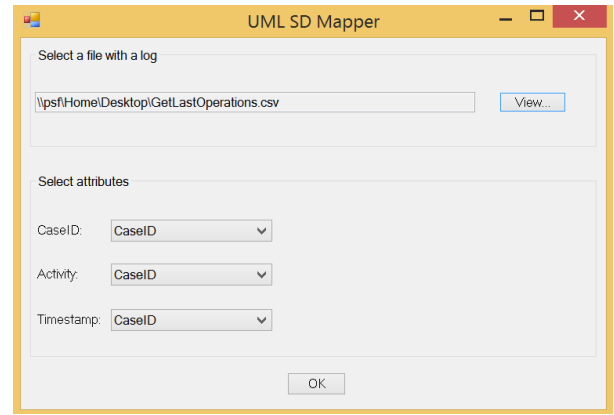


Fig. 7. Event log configuration

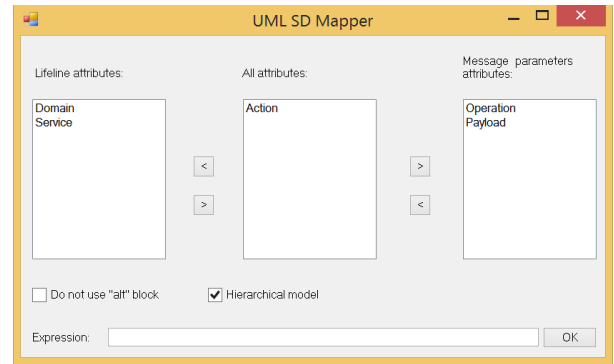


Fig. 8. Diagram configuration

a direction of our future work. UML sequence diagrams do not always show parallel interactions properly. Thus, we are going to mine hybrid diagrams as UML sequence diagrams with a *ref* fragment, which abstracts parallel interactions and refers to UML activity diagram illustrated parallel processes.

## ACKNOWLEDGEMENT

This work is supported by the Basic Research Program at the National Research University Higher School of Economics and Russian Foundation for Basic Research, project No. 15-37-21103.

## REFERENCES

- [1] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011.
- [2] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, Incorporated, 1st edition, 2011.
- [3] Rubin V.A. Shershakov S.A. System runs analysis with process mining. In *Modeling and Analysis of Information Systems*, pages 818–833, 2015.
- [4] Atanas Rountev and Beth Harkness Connell. Object naming analysis for reverse-engineered sequence diagrams. In *Proceedings of the 27th International Conference on Software Engineering, ICSE '05*, pages 254–263, New York, NY, USA, 2005. ACM.
- [5] Atanas Rountev. Static control-flow analysis for reverse engineering of uml sequence diagrams. In *In Proc. 6th Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, pages 96–102. ACM Press, 2005.

- [6] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from c++ code. pages 159–168. IEEE Computer Society, 2003.
- [7] E. Korshunova, Marija Petkovic, M. G. J. van den Brand, and Mohammad Reza Mousavi. Cpp2xmi: Reverse engineering of uml class, sequence, and activity diagrams from c++ source code. In *WCRE*, pages 297–298. IEEE Computer Society, 2006.
- [8] Sparx Systems’ Enterprise Architect. <http://www.sparxsystems.com.au/products/ea/>.
- [9] IBM Rational Software Architect. <https://www.ibm.com/developerworks/downloads/r/architect/>.
- [10] Visual Paradigm. <https://www.visual-paradigm.com/features/>.
- [11] Altova UModel. <http://www.altova.com/umodel.html>.
- [12] MagicDraw. <http://www.nomagic.com/products/magicdraw.html>.
- [13] StarUML. <http://staruml.io>.
- [14] ArgoUML. <http://argouml.tigris.org>.
- [15] Hafeez Osman and Michel R. V. Chaudron. Correctness and completeness of CASE tools in reverse engineering source code into UML model. *The GSTF Journal on Computing (JoC)*, 2(1), 2012.
- [16] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Trans. Softw. Eng.*, 32(9):642–663, September 2006.
- [17] Romain Delamare, Benoit Baudry, and Yves Le Traon. Reverse-engineering of uml 2.0 sequence diagrams from execution traces. In *Proceedings of the workshop on Object-Oriented Reengineering at ECOOP 06*, Nantes, France, July 2006.
- [18] Tewfik Ziadi, Marcos Aurélio Almeida da Silva, Lom-Messan Hillah, and Mikal Ziane. A fully dynamic approach to the reverse engineering of uml sequence diagrams. In Isabelle Perseil, Karin Breitman, and Roy Sterritt, editors, *ICECCS*, pages 107–116. IEEE Computer Society, 2011.
- [19] Yann gaël Guéhéneuc. Automated reverse-engineering of uml v2.0 dynamic models. In *Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering*. <http://smallwiki.unibe.ch/WOOR>, 2005.
- [20] OMG. The OMG and Service Oriented Architecture, 2006.
- [21] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.
- [22] Visio 2010: Software Development Kit, 2010. <https://www.microsoft.com/en-us/download/details.aspx?id=12365>.
- [23] Christian W. Günther and Eric Verbeek. XES Standart Definition version 2.0, 2014.
- [24] Scott Owens, John Reppy, and Aaron Turon. Vtmime framework as applied to process mining modeling. pages 166–179, 2015.

# Applying MapReduce to Conformance Checking

Ivan Shugurov, Alexey Mitsyuk

National Research University Higher School of Economics,  
Laboratory of Process-Aware Information Systems,  
20 Myasnitskaya St., Moscow 101000, Russia  
Email: shugurov94@gmail.com, amitsyuk@hse.ru

**Abstract**—Process mining is a relatively new research field, offering methods of business processes analysis, which are based on their execution history (event logs). Conformance checking is one of the main sub-fields of process mining. Conformance checking algorithms are aimed to assess how well a process model and an event log correspond to each other. The paper deals with the problem of high computational complexity of the alignment-based conformance checking algorithm. This particular issue is important for checking the conformance between models and real-life event logs, which is quite problematic using existing approaches.

MapReduce is a popular model of parallel computing which allows simple implementation of efficient and scalable distributed calculations. In this paper, a MapReduce version of the alignment-based conformance checking algorithm is described and evaluated. We show that conformance checking can be distributed using MapReduce and that computation time scales linearly with the growth of size of event logs.

**Index Terms**—Process mining, Conformance checking, MapReduce, Hadoop, Big data.

## I. INTRODUCTION

Ever-increasing size and complexity of modern information systems force both researchers and practitioners to find novel approaches of formal specification, modeling, and verification. This process is absolutely essential for ensuring their robustness and for possible optimization and improvements of existing business processes. Process mining is a research field which offers such approaches [1]. *Process mining* is a discipline which combines techniques from data analysis, data mining and conventional process modeling. Typically, three main sub-fields of process mining are distinguished in the literature: (1) process discovery; (2) conformance checking and (3) enhancement [1].

The aim of *process discovery* is to build a process model based solely on the execution history of a particular process. Event logs are the most common and natural way of persisting and representing execution history. By an event log we understand a set of traces where each trace corresponds exactly to one process execution. A typical process discovery algorithm takes an event log as an input parameter and constructs a process model which adequately describes the behavior observed in the event log.

The task of *conformance checking* is to measure how well a given process model and an event log fit each other. Furthermore, usually showing only the coefficient of conformance is insufficient for real-life application since analysts often need to see where and how often deviations happen in

order to draw any conclusions. Therefore, it is often the case when conformance checking algorithms include computation of additional metrics as well as visualization of deviations.

*Process enhancement* deals with improvements of processes as well as corresponding process models.

One of the challenges of process mining, when applied in real life, is the size of data to be processed and analyzed [2], [3]. Since process discovery has drawn significant attention of researchers, there are a number of solutions which allow fast process discovery from large event logs [4]. These solutions vary from using distributed systems and parallel computing [5] to applying more efficient algorithms, which require less data scans and manipulations [6], [7]. In contrast, conformance checking remains problematic to be made fast due to its theoretical and algorithmic difficulties. At the same time, efficient, easy-to-use and robust conformance checking is the key to better process improvement since enhancement approaches often rely heavily on measuring conformance (for example, see model repair approaches [8], [9]).

This paper is focused on implementation details of distributed conformance checking rather than on its theoretical aspects. It describes a possible way of speeding up conformance checking. It implies improving one of the existing conformance checking algorithms so that it can be executed in a distributed manner by means of using MapReduce [10]. One of the very first papers discussing distributed conformance checking [11] was dedicated solely to theoretical foundations of process models and event logs decomposition. The author takes a look at the algorithmic side of distributed conformance checking and totally skips problems of its software implementation. In this paper we consider practical aspects of distributed conformance checking. Furthermore, we prove viability of the proposed approach by demonstrating that it really allows measuring conformance of bigger event logs better than currently existing approaches.

This paper is structured as follows. Section II introduces foundational concepts we use in the paper. In section III, the reader can find the main contribution. Section IV proposes several improvements of the approach proposed in section III. An implementation of the presented approach is described in section V. Related work is reviewed in section VI. Finally, section VII concludes the paper.



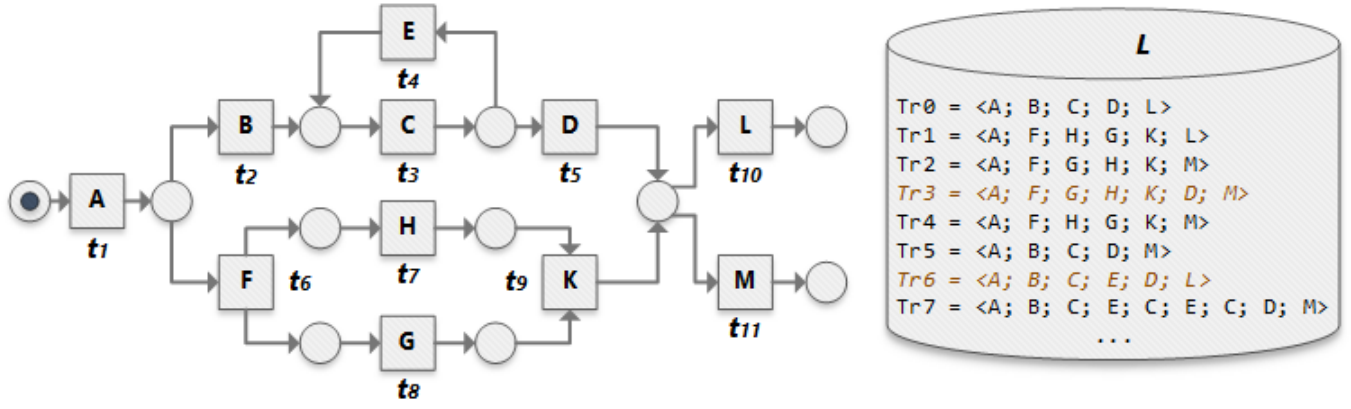


Fig. 1. Petri Net and Event Log

## II. PRELIMINARIES

In this paper we consider process models in the Petri net (simple P/T-nets) notation. A *Petri net* is a bipartite graph which consists of nodes of two types. In process mining, transitions, denoted by rectangles, are considered as process activities, whereas places, denoted by circles, designate the constraints imposed on the control-flow. String labels may be associated with transitions in order to show the correspondence between activities and transitions. Transitions without labels are called *silent*. It implies that silent transitions model behavior and constraints of an activity in a process, executions of which are not recorded into event logs. Each place denotes a causal dependence between two or more transitions. Places may contain so-called *tokens*. A transition may fire if there are tokens in all places connected to it via incoming arcs. When fired, it consumes one token from each input place and produces one token to each output place. *Marking* is a distribution of tokens over all places of a Petri net, thus a marking denotes the current state of a process.

An *event log* is a recorded history of process runs. Usually the execution of a process in some information system is recorded for documenting, administrative, security, and other purposes. The main goal of process mining is to explore and these data for the diagnosis and improvement of actual processes.

We consider event logs of standardized nature as they are used in process mining. Formally, an event log is multiset of traces where each trace is a sequence of events. Each trace corresponds to exactly one process run. An event contains the name of associated activity, timestamp, performer name and may contain other additional properties. In this paper we consider simple event logs, in which events contains only names of activities. An example model with the corresponding event log are shown in Figure 1.

### A. Conformance Checking

The conformance checking and its place in process mining are defined in [1]. Usually four dimensions of conformance are considered: fitness, precision, generalization, and simplicity.

However, this paper focuses exclusively on fitness. By the term *fitness* we understand the extent to which a model can reproduce traces from an event log. In other words, fitness shows how well the model reflects the reality. The fitness dimension is typically regarded as being the most frequently used and best-defined [1] among the dimensions.

Nowadays, the most advanced and refined conformance checking approach is the one using alignments [12]. The term *alignment* is used to denote the set of pairs where each pair consists of an event from an event log and a corresponding transition of a model. Such pairs are constructed sequentially for each event in a trace. A simple alignment for the trace  $Tr3$  (see Figure 1) is depicted in Figure 2. However, it is allowed to pair an event with no transitions (special "no move" symbol  $\gg$ ). This means that the event is present in a log but cannot be replayed by any transition in the model. It is also possible to map a transition to no events (this is denoted by the same symbol  $\gg$ ). In that case the transition is fired but there are no evidence of this fact in the event log. Thus, there are two main types of steps composing any alignment: a synchronous move (a transition fired with the same label as an event name from the event log) and a non-synchronous move (a transition label and an event name are the different ones, or a move is skipped either in the model or in log).

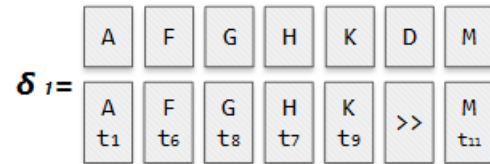


Fig. 2. Alignment

Alignments help to measure the difference between a trace from an event log and behavior specified by a model. In order to quantify the difference one has to calculate the number of non-synchronous moves and assess their significance. This assessment is accomplished by introducing a *cost function*, which is used for calculating *cost of an alignment*. By *cost*

we understand a number which somehow designates the significance. The general idea is that some deviations are more severe than others, thus these deviations have more impact on the overall conformance. Using cost function one can assign cost for each type of deviation for each transition and event. Thus, cost function maps a pair of an event and a transition to a number, which signifies a penalty for having such a pair in a trace. The more the cost is, the more significant this deviation is. Assuming that all costs are set to 1, the alignment shown in Figure 2 has the cost 1, because there is only one non-synchronous move in it (event  $D$  in the trace has to be skipped during model run). Accumulating costs for all alignments of a particular event log, it is possible to derive the cost for the entire log.

It is possible that a particular run through the model and a particular trace have several possible alignments. In order to choose between them a cost function is used to evaluate the cost of each alignment. An alignment with the lowest cost is selected as the *optimal alignment*. According to [12] it makes sense to use only optimal alignments when calculating fitness. Alignment-based fitness can be measured using the metric defined in [13]:

$$f(L, N) = 1 - \frac{\sum_{tr \in L} \sum_{e \in tr} cost_{fn}^{\delta_{opt}}(e, N)}{\sum_{tr \in L} cost_{ai}}, \quad (1)$$

where  $L$  is an event log,  $N$  is a model,  $cost_{fn}^{\delta_{opt}}(e, N)$  is a cost of a pair  $(e, (t_i, t_i^l))$  ( $e$  is an event,  $t_i$  is a transition from model run,  $t_i^l$  is its label) in the particular optimal alignment  $\delta_{opt}$ , which depends on used cost function  $cf$ ,  $cost_{ai}$  is a total cost of the trace  $tr$  if all moves in it are considered as non-synchronous. Thus, fitness is a normalized ratio of the accumulated costs calculated for the optimal alignments to the accumulated costs for the worst possible alignments for a particular event log.

It is shown in [12] that construction of alignments and selection of optimal among them for each trace can be converted to solving the shortest path problem. Formally, a trace from the event log is represented as an *event net*, which is a special Petri net having the form of the sequence of transitions connected through places. Then the product of the model and this event net is constructed. It is shown in [12] that the problem of optimal alignment calculation can be viewed as a problem of finding a firing sequence in this product, which can be achieved by using a state-space exploration approach.

The proposed approach has a low computational performance when dealing with large models, large event logs or in case of low fitness because of the necessity to solve the shortest path problem, especially for model of certain types [12]. The author himself states in [12] that "from a computational point of view, computing alignments is extremely expensive". Moreover, its existing implementation keeps the processed models, event logs, event nets, and computed alignments in computer's main memory. This approach allows for flexible configuration of visualization settings, and, in some cases, faster completion. However, this feature makes usage of ex-

isting implementation rather hard and inconvenient because the algorithm typically consumes several gigabytes of main memory even for processing relatively small models and small event logs (dozens of megabytes). Thus, it is not suitable for real-life usage.

This paper proposes a way of checking conformance between process models and big event logs of gigabyte sizes using MapReduce.

## B. MapReduce

*MapReduce* is a computational model proposed and popularized in [10], although the idea dates back to the origins of functional programming. MapReduce is a popular technology among practitioners and a research area among scientists. It has a good tool support, all major cloud platform vendors provide the possibility to execute MapReduce jobs on their cloud clusters.

The model simplifies parallel and distributed computing by allowing software developers to define only two quite primitive functions: *map* and *reduce*. At each invocation of a map function (also called *mapper*), it takes a key-value pair and produces an arbitrary number of key-value pairs. The aim of reduce functions (also called *reducers*) is to aggregate values with the same key and perform necessary computations over them. Thus, a reduce function takes a key-list pair as input parameters. Usage of such rather trivial functions makes their distribution straightforward. Last but not least, comes another important function allowed by MapReduce which is called *combine*. Its main purpose is to perform reduce-like computation between mappers and reducers. Combine functions (also known as *combiners*) are invoked on the same very computers as mappers. Combiners allow for further parallelizing computations and decreasing amount of data transferred to reducers and processed by them. It was pointed out even in the original article [10] that combiners may dramatically decrease computation time.

One of the most crucial advantages of MapReduce is that algorithms expressed in such a model are inherently deadlock-free and parallel. Another important advantage is the tendency to perform computations where required data resides. Generally, computation of map tasks take place where the required data is stored since its location is known beforehand. Such an approach ensures that data transfer between computers and latency, inflicted by it, are minimized. Ideally, data is transferred between computers where map tasks are executed and computers where reduce tasks are executed. Unfortunately, it is rarely achievable since all files are separated into smaller parts, called *blocks*, and distributed (and also replicated) over a cluster, thus data needed for execution of a single map task may reside in different data chunks — there will be a need to move a portion of data from one computer to another.

## III. FITNESS MEASUREMENT USING MAPREDUCE

This section describes the approach we propose for checking conformance.



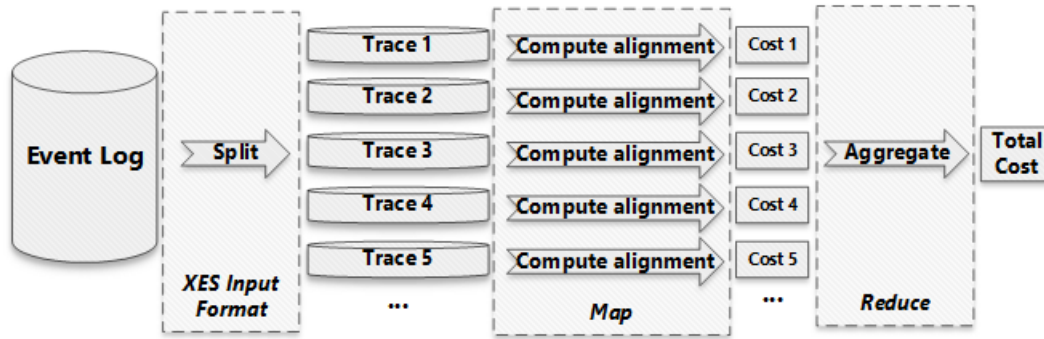


Fig. 3. Conformance Checking with MapReduce

The few adjustments of the existing conformance checking algorithm with alignments need to be done in order to implement the proposed schema. It is expected that the algorithm will benefit if distribution is applied to traces. It means that traces are distributed over a cluster so that their alignments can be computed in parallel. Another possible option was to distribute computation of each alignment since efficient distributed graph algorithms for solving the shortest path problem are known. However, use of them seems excessive because they are aimed at solving problems on graphs consisting of thousands and millions of nodes, which is not the case for business process models. A process model consisting of more than a hundred nodes seems unrealistic.

The general schema is depicted in Figure 3. *Map function* takes traces one by one and computes their alignments. This process can easily be carried out in parallel since, by its definition, an alignment is computed individually for each trace. It is enough to use a single *reduce function* which aggregates fitnesses of all traces and calculating fitness of the overall event log. Single reducer implies that key-value pairs emitted by all mappers have the same key. Single reducer can be considered as a bottleneck due to the reason that before it can start processing it waits for completion of all maps and transition of all costs to a single computer. To diminish the negative effect of a single reducer, a combiner function comes in handy. The problem is that calculating average is not an associative operation, thus it is impossible to use the basic reduce function instead of the combine function. We implemented it in a manner resembling the one described in [14]. The general idea is that calculating average can be easily decomposed into calculating a sum of all entries of some metric and counting a number of entries, where both of them are associative operations. It implies changing the structure of values used in key-value pairs. The modified version of values contains not only statistics (fitness and so on) but also a counter which shows how many traces describes a particular value. Given that, combiners only have to sum the values they receive and increment the counter.

#### IV. POTENTIAL IMPROVEMENTS

One of the possible improvements of the algorithm is to enhance it by adding trace deduplication. When large event

logs are considered, the possibility of the equivalent traces occurring several times is very high. Hence, it might be desired to find only unique traces, number of their occurrences and compute alignments only for them. It will allow for lessening the number of computed alignments. However, efficient MapReduce algorithm for deduplication of event sequences is far from trivial. Moreover, it is not guaranteed that time needed for deduplication and subsequent conformance checking will be shorter than in case of using the standard approach. This question can only be answered by conducting relevant experiments.

Even though process models are not prone to be large, a lot of time is still required for checking conformance. Another possible improvement, which aims at reducing model size, is to employ the "divide and conquer" principle. The way how the principle can be applied to cope with high computational complexity of conformance checking was proposed in [15] and [16]. The general idea is to divide a process model into smaller sub-parts. Next step is event log projection. This means that for each fragment of a model all events from the event log that correspond (names of events are equal to labels of activities) to a particular fragment are selected. As a result, we get as many projected event logs as decomposed Petri net fragment.

Once it is done, alignments and costs of each fragment can be computed. Then it is possible to sum costs of parts following specific rules to get a lower bound of the cost of the entire log. Having these costs, an upper bound of fitness can be computed. Performance gain is the most crucial motivation of this approach. Since time needed for computing alignments depends on trace size, usage of smaller parts of the model ensures faster computation. A wide range of model decomposition strategies have been proposed in [17], [15], [18], which leaves the user with the necessity to empirically choose between them. Last but not least, decomposition also incurs time overhead and projected event logs take up disk space, so usage of the algorithm is not beneficial (or even feasible) in all the possible cases. Furthermore, there is no research done to establish when usage of which approach makes more sense.

It is possible to employ a similar approach in the MapReduce environment. There are two possible options: (1) compu-

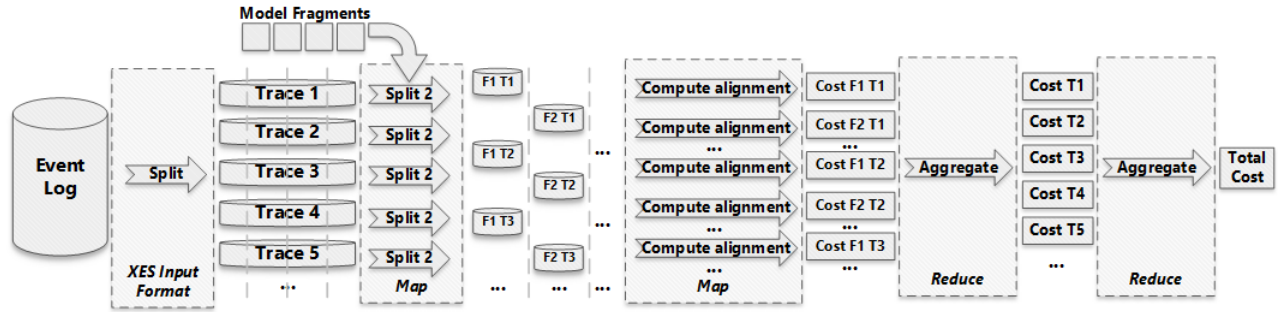


Fig. 4. Possible Approach with Vertical Decomposition

tation of the overall event log fitness and (2) computation of fitness of each separate model part. In all the cases fitness is computed in a three-stage process as it is shown in Figure 4. The zero stage again is the splitting of the log by traces, which is followed by trace decomposition. Traces are decomposed using the maximal decomposition described in [15]. However, incorporation of other decomposition techniques [15], [17], [16] is also possible. At the second stage, alignments of sub-traces are computed and then aggregated. The final stage differs depending on the selected computation option. At this stage either fitness of the overall event log is computed at a single reducer or fitnesses of individual parts are computed at different reducers (the number of reducers can be up to the number of model parts). If fitness of individual process parts is calculated, after the second map unique identification of a model part is used as a key for emitted key-value pairs. When decomposition is applied, log deduplication's importance and potential benefit grow even more.

## V. IMPLEMENTATION AND TESTING

This section describes the actual implementation<sup>1</sup> of the proposed approach and its experimental testing. *Hadoop* [19] was used for implementation and testing of the approach because it is a common and widely supported open source tool.

### A. Implementation

The original algorithm was implemented as a ProM Framework plugin. The ProM Framework [20], [21] is a well-known tool for implementation of process mining algorithms. The ProM Framework consists of two main components:

- 1) ProM core libraries which are responsible for the main functionality used by all users and extensions;
- 2) extensions (typically called plugins) which are created by researchers and are responsible for import/export operations, visualization, and actual data processing.

The platform is written in such a way that it allows plugins to use data produced by other plugins. Furthermore, ProM encourages programmers to separate concerns: export plugins are only used for exporting data, visualization plugins are used for visualizing objects. As a result, a common usage

scenario always consist of a chain of invocations of different plugins. Among main advantages of ProM are configurability, extensibility, and simplicity of usage. Last but not least, the platform allows researchers to easily create and share plugins with others thus extending the tool and contributing to the overall field of process mining. Despite all these positive sides, usage of ProM can be inconvenient and tedious, if the desired goal is unusual in any way.

XES [22] is often considered as a de facto standard for persisting event logs in the area of process mining. Technically, it is an XML-based standard, which means that it is tool-independent, extensible, and easy to use. Moreover, ProM fully supports this standard and has all required plugins for working with it.

Our approach involves usage of raw event logs stored in the format of XES only at the zero step of the algorithm. Before separate traces are available for the required computations, it is necessary to sequentially read XES files dividing them into separate traces. It is accomplished by using the *XMLInputFormat* from the *Mahout* project [23]. *XMLInputFormat* provides the capability of extracting file parts located between two specified tags. Moreover, the class is responsible for ensuring that the entire requested part (in our case — trace) is read, no matter in which blocks and on which data nodes it resides.

The fact that the initial algorithm was implemented for ProM inflicts several inconveniences for its distribution. First of all, it is assumed that the plugin is invoked by ProM via a special context. Essentially, it implies several things:

- 1) the entire ProM distribution has to be sent to each computational node;
- 2) at each computational node, it is required to start up ProM (it may take up to couple of minutes on an average computer).

As a result, it may significantly increase latency and incur higher time needed for termination of computations. To avoid this, it was decided to alter implementation in such a way that a number of libraries the algorithm depends on in as minimal as it is possible to achieve. In other words, on the one hand it was desired to separate the implementation of the algorithm from ProM. On the other hand, usage of ProM could be useful for initial settings and visualization of final results. As a result, we achieved such a level of decoupling, that it is possible to launch

<sup>1</sup>The tool is available at <https://sourceforge.net/p/distributedconformance/>

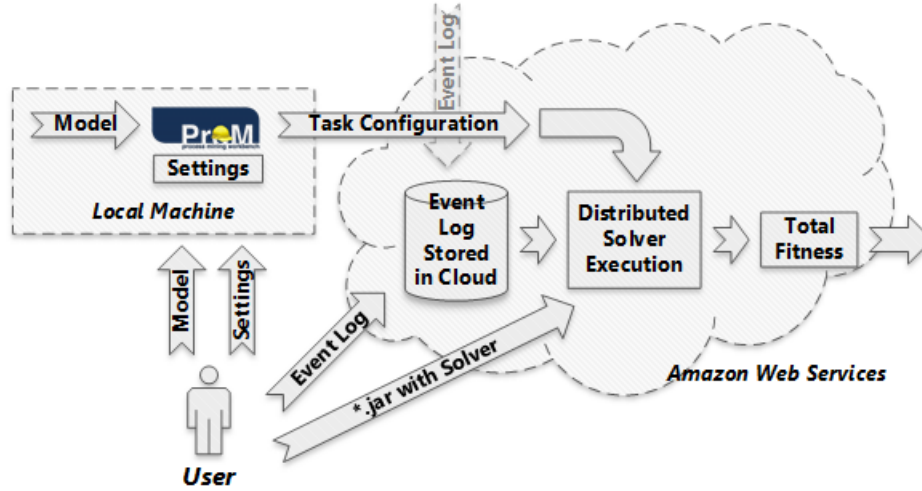


Fig. 5. Implementation of the Approach

the algorithm completely autonomously without the need of installation of the ProM Framework or any ProM plugins.

The resulting architecture is illustrated in Figure 5. Conformance measurement is done in two-step approach. At first step, the user loads a model, represented by a Petri net into a special ProM plugin which serves for setting the options of the alignments-based conformance algorithm (mapping between transitions and events in event logs, costs of insertion and skipping in alignments). We use standard ProM classes for representing Petri nets because they allow for easier compatibility with other ProM plugins. Loading a model to a main memory should not be a problem because it is highly unlikely for such models to contain even hundreds of nodes, thus the size of process models is typically relatively small. Another possible option was to specify settings exclusively via XML files, though we found it less intuitive and convenient than visual settings. Once the algorithm is configured, settings are written to a file which later will be uploaded to a cluster. Last but not least, it is important to state that this ProM plugin depends neither on Hadoop nor on a chosen cloud cluster nor on any other auxiliary Hadoop libraries.

When Hadoop job is initiated, the user is asked to specify directories where event logs are placed, a path to a Petri net, and a path to conformance settings. A model and settings are then automatically added into the Hadoop *distributed cache* — the files are replicated to each data node, so they are available for fast access by any mapper. At a startup of each model, the files are loaded into main memory because they will be used for all the alignment computations.

After completion of conformance measurement, the results are written to a single file which afterwards can be downloaded and viewed in ProM. Another sub-task is to find in which cases deduplication is worthwhile and how exactly it affects computational time.



Fig. 6. Computation time of the standard approach

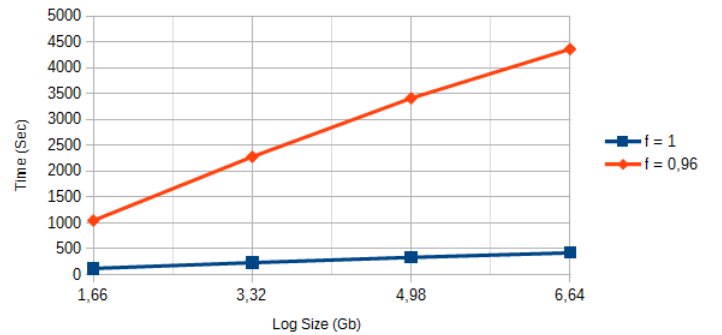


Fig. 7. Computation time with MapReduce

## B. Experimental Results

The proposed algorithm was tested and evaluated using Amazon Web Services [24]. In our cluster we used five *m3.xlarge* instances (one as a master node, four as data nodes). A local computer used for conducting experiments with the original algorithm had the following configuration: Intel Core i7-3630QM, 2.40 GHz, 8 GB of main memory, Windows 7 64 bit.

For testing purposes we created a process model comprising some of the main workflow patterns: sequence, parallel split,

synchronize, exclusive choice, and simple merge [25]. Afterwards, several models derived from the original were created — they all differ in fitness. Artificial event logs were generated using the approach proposed in [26]. Logs were generated only for the original model. All resulting logs were of different sizes.

Figure 6 illustrates how computation time depends on a number on traces and fitness. It is clear from the plots that computational complexity scales linearly with the growth of a number of traces. Moreover, it is seen that computation time highly depends on fitness. The lower the fitness, the slower the computations will be. It seems that computation time does not scale linearly with the decrease of fitness if the same quantity of logs is used. The clear indicators are the margins between lines representing fitness 1 and 0.96, and 0.96 and 0.9. Furthermore, we can conclude that the lower fitness, the faster computation time increases with the rise of the number of traces.

Figure 7 provides an overview of how the algorithm scales when it is distributed using MapReduce. It is worth mentioning that 1.66 Gb of logs contain 500 thousand traces. As in the case of the not distributed algorithm, the graph shows that the algorithm scales linearly with the increase of a number of traces. Furthermore, similarly to the not distributed case, for non-fitting models computations take considerably longer than for perfectly fitting ones, and that computation time grows faster for non-fitting models.

In Figure 8 a comparison of distributed and not distributed approaches is provided. Unfortunately, it is impossible to establish exactly when the distributed implementation beats the original in terms of performance since the original one cannot handle event logs of considerable size. In addition, the original algorithm was not able of handling more than a hundred of Mbytes. On data of such small sizes MapReduce and Hadoop fail to work efficiently because they are designed for processing much bigger files. As a matter of fact, Hadoop does not parallelize processing of files which are smaller than a single file block. It is clear from Figure 8 that for relatively small event logs the distributed version works more slowly. It is clear from the graph that our solutions can handle event logs of several dozens of GBs even on a small cluster used for conducting these experiments.

## VI. RELATED WORK

Although applicability of MapReduce or distributed systems for the tasks of process mining has not drawn significant attention yet, there are a few papers which consider this subject.

In [27] the authors focus exclusively on finding process and events correlation in large event logs. According to them, MapReduce solution for such a computationally and data intensive task as events correlation discovery performs well and can be scaled to large datasets.

Other works where the authors study applicability of MapReduce to process mining are [28], [29]. In these articles, a thorough description of several popular discovery

algorithms is provided (the alpha algorithm [30], and the flexible heuristics miner [31]). Every one of them consists of several consequent MapReduce jobs. First MapReduce jobs is responsible for reading event logs from the disc, splitting them into traces, and ordering event in each trace. The general idea of the second MapReduce all the implementations is that first step of process discovery typically requires extracting trivial dependencies between events called *log-based ordering relations*. Examples of those are:

- $a > b$  — event  $a$  is directly followed by event  $b$ ,
- $a >> b$  — a loop of length two,
- $a >>> b$  — event  $a$  is followed by event  $b$  somewhere in the log.

These relations can be found individually for each trace. Therefore, their computations are trivially parallelized using Mappers. Further MapReduce jobs vary but they somehow use mined primitive log-ordering relations to build a process model. The main potential problem of implementations is that these further MapReduce jobs typically compute relations for the overall event log. To achieve this, it is often the case when it is necessary for mappers to produce identical keys for all emitted pairs so that they all end up on the same computer and processed by the same reducer. Moreover, the proposed implementations extensively use *identity mappers*. It is a standard term for mappers which emit exactly the same key-value pairs as they receive without performing any additional computations — all useful computations performed by combiners or reducers. They are used only because MapReduce paradigm requires presence of mappers. Despite these concerns, it is shown that performance and scalability provided by MapReduce are good enough for the task of process discovery from large volumes of data. Our solution, in contrast to the described above, uses a more suitable file format. It allows measuring conformance without extra steps needed for preliminary log transformations.

In [32] the authors describe their framework for simplified execution of process mining algorithms on Hadoop clusters. The primarily focus of this work is to show how process mining algorithm can be submitted to a Hadoop cluster via the ProM user interface. In order to demonstrate viability of their approach, the authors claim that they implemented and tested the Alpha miner, the flexible heuristics miner, and the inductive miner [33]. We opted for not using the presented framework in order to simplify the usage of our ProM plugin and not to force the user to download all the codebase required by Hadoop and its ecosystem.

To sum up, these papers clearly demonstrate not only that process mining can benefit from using distributed systems and MapReduce, but also that such distributed process mining algorithms are needed and desired for usage in the real-life environment. Moreover, from these papers it is clear that some common approaches and techniques of process mining suit the MapReduce model well. Last but not least, analysis of the related work reveal that there are only theoretical considerations of parallel or distributed conformance checking

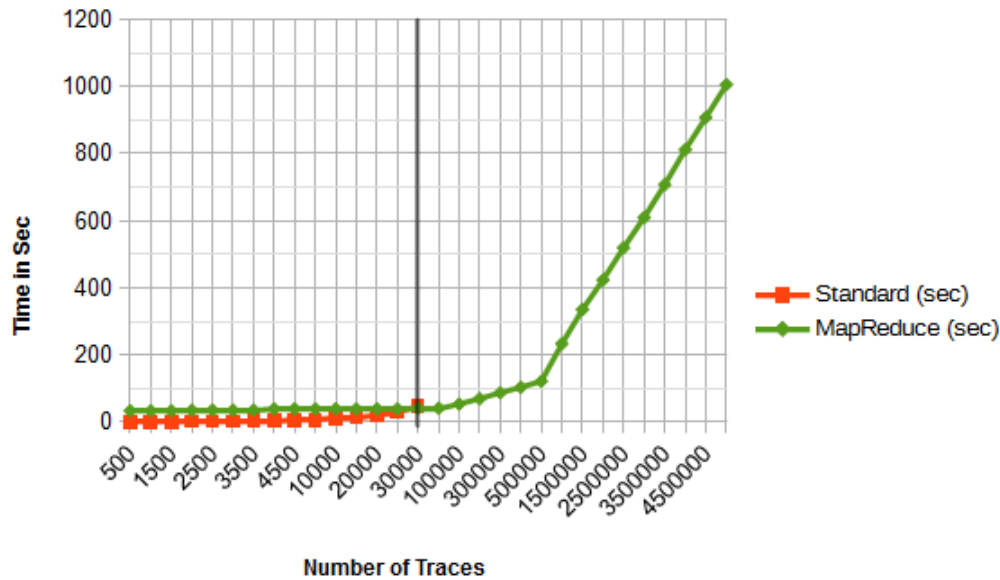


Fig. 8. Comparison of the standard and the distributed approaches

and its usefulness.

## VII. CONCLUSIONS

This paper presents one of the possible ways of speeding up large-scale conformance checking. The paper provides a helicopter-view of distributed conformance checking and suggests ways for possible extensions and improvements. One of the proposed algorithms was implemented and evaluated on event logs which were different in terms of size and fitness.

As a possible extension it is worth considering implementing the algorithm using the *Spark* framework rather than Hadoop because as it is often claimed Spark might provide better performance due to its in-memory nature. Furthermore, the XES standard which defines how event logs should be structured for convenient process mining, but it seems that the XES standard is not the best option for using with Hadoop. Thus, it is possible to consider other storage formats such as Hadoop sequence files or the *Avro* format.

## ACKNOWLEDGMENT

This work is supported by the Basic Research Program at the National Research University Higher School of Economics and Russian Foundation for Basic Research, project No. 15-37-21103.

## REFERENCES

- [1] Wil M. P. van der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011.
- [2] S. A. Shershakov and V. A. Rubin, "System runs analysis with process mining," *Modeling and Analysis of Information Systems*, vol. 22, no. 6, pp. 818–833, December 2015.
- [3] S. A. Shershakov, "VTMine framework as applied to process mining modeling," *International Journal of Computer and Communication Engineering*, vol. 4, no. 3, pp. 166–179, May 2015.
- [4] W. M. van der Aalst, "Process Mining in the Large: A Tutorial," in *Business Intelligence*. Springer, 2014, pp. 33–76.
- [5] C. Bratosin, N. Sidorova, and W. van der Aalst, "Distributed Genetic Process Mining," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, 2010, pp. 1–8.
- [6] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour," in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing, N. Lohmann, M. Song, and P. Wohed, Eds. Springer International Publishing, 2014, vol. 171, pp. 66–78.
- [7] A. A. Kalenkova, I. A. Lomazova, and W. M. P. van der Aalst, "Process Model Discovery: A Method Based on Transition System Decomposition," in *Petri Nets*, ser. Lecture Notes in Computer Science, vol. 8489. Springer, 2014, pp. 71–90.
- [8] D. Fahland and W. M. P. van der Aalst, "Model Repair - Aligning Process Models to Reality," *Inf. Syst.*, vol. 47, pp. 220–243, 2015. [Online]. Available: <http://dx.doi.org/10.1016/j.is.2013.12.007>
- [9] I. S. Shugurov and A. A. Mitsyuk, "Iskra: A Tool for Process Model Repair," *Proceedings of the Institute for System Programming*, vol. 27, no. 3, pp. 237–254, 2015.
- [10] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Online]. Available: <http://doi.acm.org/10.1145/1327452.1327492>
- [11] W. M. P. van der Aalst, "Distributed Process Discovery and Conformance Checking," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, J. de Lara and A. Zisman, Eds. Springer Berlin Heidelberg, 2012, vol. 7212, pp. 1–25.
- [12] A. Adriansyah, "Aligning Observed and Modeled Behavior," PhD Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 2014.
- [13] A. Adriansyah, B. van Dongen, and W. M. van der Aalst, "Conformance Checking using Cost-Based Fitness Analysis," in *IEEE International Enterprise Computing Conference (EDOC 2011)*, C. Chi and P. Johnson, Eds. IEEE Computer Society, 2011, pp. 55–64.
- [14] D. Miner and A. Shook, *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*, 1st ed. O'Reilly Media, Inc., 2012.
- [15] W. M. P. van der Aalst, "Decomposing Petri Nets for Process Mining: A Generic Approach," *Distributed and Parallel Databases*, vol. 31, no. 4, pp. 471–507, 2013.
- [16] J. Munoz-Gama, "Conformance checking and diagnosis in process mining," PhD Thesis, Universitat Politècnica de Catalunya, 2014.
- [17] W. M. P. van der Aalst, "Decomposing Process Mining Problems Using Passages," in *Application and Theory of Petri Nets*, ser. Lecture Notes



- in Computer Science, S. Haddad and L. Pomello, Eds. Springer Berlin Heidelberg, 2012, vol. 7347, pp. 72–91.
- [18] J. Munoz-Gama, J. Carmona, and W. M. van der Aalst, “Single-Entry Single-Exit Decomposed Conformance Checking,” *Information Systems*, vol. 46, pp. 102–122, 2014.
  - [19] “Apache hadoop,” <http://hadoop.apache.org/>, accessed: 2016-04-01.
  - [20] W. M. P. van der Aalst and B. van Dongen, C. Günther, A. Rozinat, E. Verbeek, and T. Weijters, “ProM: The Process Mining Toolkit,” in *Business Process Management Demonstration Track (BPMDemos 2009)*, ser. CEUR Workshop Proceedings, A. Medeiros and B. Weber, Eds., vol. 489. CEUR-WS.org, 2009, pp. 1–4.
  - [21] “Prom framework,” <http://www.promtools.org/doku.php>, accessed: 2016-04-01.
  - [22] IEEE Task Force on Process Mining, “XES Standard Definition,” [www.xes-standard.org](http://www.xes-standard.org), 2013.
  - [23] “Apache mahout,” <http://mahout.apache.org/>, accessed: 2016-04-01.
  - [24] “Amazon EMR,” <https://aws.amazon.com/ru/elasticmapreduce/>, accessed: 2016-04-01.
  - [25] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, “Workflow Patterns,” *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003. [Online]. Available: <http://dx.doi.org/10.1023/A:1022883727209>
  - [26] I. S. Shugurov and A. A. Mitsyuk, “Generation of a Set of Event Logs with Noise,” in *Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2014)*, 2014, pp. 88–95.
  - [27] H. Reguieg, F. Toumani, H. R. Motahari-Nezhad, and B. Benatallah, “Using MapReduce to Scale Events Correlation Discovery for Business Processes Mining,” in *Business Process Management*. Springer, 2012, pp. 279–284.
  - [28] J. Evermann, “Scalable Process Discovery using Map-Reduce,” *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, 2014.
  - [29] J. Evermann and G. Assadipour, “Big Data meets Process Mining: Implementing the Alpha Algorithm with Map-Reduce,” in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1414–1416.
  - [30] W. M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster, “Workflow Mining: Discovering Process Models from Event Logs,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.
  - [31] A. J. M. M. Weijters and J. T. S. Ribeiro, “Flexible Heuristics Miner (FHM),” in *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on*, April 2011, pp. 310–317.
  - [32] S. Hernandez, S. Zelst, J. Ezpeleta, and W. M. P. van der Aalst, “Handling big (ger) logs: Connecting ProM 6 to Apache Hadoop,” in *Proceedings of the BPM2015 Demo Session, ser. CEUR Workshop Proceedings*, vol. 1418, 2015, pp. 80–84.
  - [33] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, “Discovering Block-Structured Process Models from Incomplete Event Logs,” in *Application and Theory of Petri Nets and Concurrency*, ser. Lecture Notes in Computer Science, G. Ciardo and E. Kindler, Eds. Springer

# Modelling the People Recognition Pipeline in Access Control Systems

Frederik Gossen\*, Tiziana Margaria<sup>†</sup>  
Lero - The Irish Software Research Centre

University of Limerick, Ireland

Email: \*frederik.gossen@lero.ie, <sup>†</sup>tiziana.margaria@lero.ie

Thomas Göke  
SysTeam GmbH

Dortmund, Germany

Email: thomas.goeke@systeambh.com

**Abstract**—We present three generations of prototypes for a contactless admission control system that recognizes people from visual features while they walk towards the sensor. The system is meant to require as little interaction as possible to improve the aspect of comfort for its users. Especially for people with impairments, such a system can make a major difference. For data acquisition, we use the Microsoft Kinect 2, a low-cost depth sensor, and its SDK. We extract comprehensible geometric features and apply aggregation methods over a sequence of consecutive frames to obtain a compact and characteristic representation for each individual approaching the sensor.

All three prototypes implement a data processing pipeline that transforms the acquired sensor data into a compact and characteristic representation through a sequence of small data transformations. Every single transformation takes one or more of the previously computed representations as input and computes a new representation from them.

In the example models presented in this paper, we are focusing on the generation of frontal view images of peoples' faces which is part of the processing pipeline of our newest prototype. These frontal view images can be obtained from colour, infrared and depth data by rendering the scene from a changed viewport. This pipeline can be modelled considering the data flow between data transformations only. We show how the prototypes can be modelled using modelling frameworks and tools such as Cinco or the Cinco-Product Dime. The tools allow for modelling the data flow of the data processing pipeline in an intuitive way.

**Index Terms**—Visual Modelling, Face Recognition, People Recognition, Computer Vision.

## I. INTRODUCTION

When using today's admission control systems some kind of interaction is required to check permission for every individual. Among the most widely used technologies are RFID chips on check cards. When attempting to pass the admission control system people have to swipe their check card through a reader to transfer a unique ID to the system. The system will then check whether or not access should be granted. Once a person is identified, it is easy to assign different levels of permission to different people. This might be useful to restrict access to certain areas in a building. Other methods for identification include PINs, passwords or keys. All these methods have one thing in common: They require the user to carry something, either physically or in mind. That means it is likely that someone who is allowed to pass the admission control system is not able to do so because he or she has forgotten his or her password or key. To overcome this issue

iris recognition, fingerprint recognition and face recognition can be used [1]. All of these methods identify a person by something that cannot be forgotten such as the eye or the face.

In our work, we focus on identity recognition from colour and depth data using low-cost depth sensors such as the Microsoft Kinect 2 [2]. These sensors offer colour, infrared and depth images at high frame rates. Our goal is to recognize people with as little interaction as possible. The user should be able to walk towards the admission control system looking forwards as he or she walks. The system picks up his or her head and face and predicts the identity that is most likely to have caused the observation. Moreover, the system will have notion of certainty. In cases where the prediction is possibly wrong a fall back method for identification will be used. This can be a PIN, password or a check card but it is also possible to redirect the person to a staff member to be identified with human capabilities.

The proposed system primarily improves the aspect of comfort for everybody who uses the admission control system as they no longer have to carry check cards or keys or remember PINs or passwords. Such an admission control system can be used in many places. Starting from fitness studios, spa and swimming pools where members have to be recognized to give access, reaching to institutions where staff members have to be recognized. In these scenarios, the proposed system primarily improves the aspect of comfort. However, in some cases people are not able to use any of the alternative methods for identification. Especially in places like hospitals and retirement homes where many people suffer from impairments such a system can make a major difference. People with Parkinson's disease might be unable to swipe a check card with the tremor in way that allows the system to read the card. They will also have problems to enter a PIN or a password while a visual recognition system would not require them to interact in a particular way. Other examples include patients with Alzheimer's disease, people wearing a cast or doctors with sterilized hands.

We are currently working on the third version of a prototype for contactless admission control. Previous versions have suggested that geometric features can contribute to reliable recognition of individuals but are alone not sufficient for reliable access control [3]. We are currently focusing on



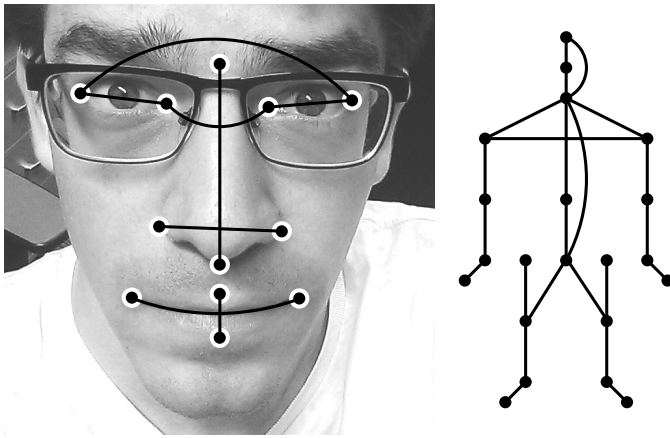


Fig. 1. Visualization of 8 distances that were extracted from the Kinect's face model (left) and 23 distances that were extracted from the Kinect's skeleton model (right). The distances are estimated in space rather than in the image plane to be invariant under the viewport.

the generation of frontal view images from people's faces which we will use to extract comprehensible and characteristic features for individuals. This will allow for recognizing them in the application of an admission control system. In what follows we will present the three versions of our prototype in Section II. All three prototypes implement a data processing pipeline that transforms the acquired sensor data into a compact and characteristic feature representation. In order to show how this pipeline can be modelled, Section III introduces the reader to the meta modelling framework Cinco and to Cinco-Products that we use to model the pipeline. We present two alternative models of the data processing pipeline in Section IV that are based on these Cinco-Products. Section V concludes this paper and points our directions of future work.

## II. PROTOTYPES

We are currently working on the third version of our prototype. The first two versions were based on the Microsoft Kinect [2] and its SDK while our new version will be based only on its low-level API that is similar to that of comparable sensors. The low-cost sensor provides capabilities to acquire colour, depth and infrared images at high frame rates. It comes with a powerful SDK that provides reliable algorithms to detect people's skeletons and faces.

In this section, we will describe the three versions of our prototype and we analyse their differences.

### A. First Prototype

Starting with the first prototype, we decided to use the Microsoft Kinect 2 sensor. This sensor acquires colour, infrared and depth images at high frame rates. Moreover, the sensor comes with a Software Development Kit (SDK) that offers a high resolution face model and a skeleton model. The first system is based on the capabilities of the Kinect SDK. We use the high resolution face model to extract characteristic geometric features with clear interpretation. The features are extracted on frame by frame basis. The compact

and comprehensible set of features is used to predict the person who is most likely to have caused the observation. In this first approach we use our own implementation of a Bayesian Classifier to perform this task on every frame separately.

As we aim to recognize human identities in a comprehensible way we need features that provide clear interpretation. As one of the first features that were used for facial recognition, geometric features fulfil this requirement [4] [5]. In contrast to early approaches we extract distances in space rather than in the image plane using the Kinect's face and its skeleton model.

The Kinect's face model provides a set of 1347 feature points many of which are interpolated. In order to obtain features with clear meaning we decided to focus on a subset of 12 feature points as shown in Figure 1. These feature points represent the eye corners, the mouth corners, the lower and the upper lip and the left, right, lower and upper boundary of the nose. The face model provides these points' positions in both, the image plane and in space. As distances in the image plane are affected by the view point we use the Euclidean distances between points in space. We extract the following distances from the face model

- the inner and the outer eye distance
- the width and the height of the nose
- the width and the height of the mouth
- the width of the left and of the right eye.

Figure 1 visualizes all of the 8 facial features. Note, that some of these might vary a lot. For instance, the estimated height of the mouth will vary when people open or close their mouth. The features are therefore not invariant to facial expressions but can nevertheless characterize the face of an individual. Note, that we extract only a small subset of possible distances with clear interpretation that we expect to be characteristic for the human face. In this way we maintain comprehensibility of our representation. Moreover, too many features would lead to overfitting during the learning process as our data set was small at this stage of the development process.

The Bayesian Classifier that we used for classification assumes conditional independence of features. This assumption is obviously violated for the proposed features meaning that the Bayesian Classifier is no longer guaranteed to be optimal. However, Bayesian Classifier are often successful although their assumptions might be violated. In order to allow for a good representation of the conditional probability distribution, we introduce another assumption that the conditional probability for each features is normally distributed. Hence, the learned model for a person can be represented by mean and variance per feature. Note, that a normal distribution is a reasonable assumption for geometric features from the Kinect face model as shown in [3]. However, this introduces yet another assumption that might be violated to some degree. The Bayesian Classifier is therefore no longer guaranteed to be optimal. However, it shows reasonable performance in many classification problems as well as in our preliminary evaluation in [3].

With recognition rates of up to 80% on a preliminary data set with only 5 people, this first prototype was far from being sufficiently accurate for reliable access control. However, it proofed, that geometric features from the Kinect's models can be used to recognize people. This first system was not exploiting the redundancy of the records among consecutive frames as its prediction was on a frame by frame basis. Moreover, the feature set was extremely small.

### *B. Second Prototype*

To overcome the weaknesses of our first prototype, we introduce more features and feature aggregation in the second version of our prototype and tested different classifiers to analyse the quality of the feature set. One feature that is expected to be particularly predictive for a person's identity is his or her height. The height varies a lot between different individuals and can be estimated using the Kinect's skeleton model. The model provides the position for 25 joints in both, the image plane and in space. We extract Euclidean distances in the same way as we extract them from the face model. In order to capture meaningful features from the model we consider distances from a selection of adjacent skeleton joints. In addition, we extract features between joints that are not adjacent if we expected the feature to be characteristic for a person. In particular, we wanted to represent a person's height and his or her shoulder width. Figure 1 shows all of the proposed 23 features that are considered from the skeleton model.

We introduced feature aggregation to this version of our prototype. Frames are still processed separately to extract the set of features from them leading to 31 values per frame. When a person approaches the Kinect sensor, up to 15 frames were considered during our experiments. As the Kinect's models have high computational demands, the low frame rate did not allow for more records in most situations. All of the 15 records aim to measure the same set of distances. In order to benefit from this redundancy, we aggregated the sequence into a single value per feature using one of 6 aggregation methods. As the most prominent aggregation method for real values, we used the mean in our experiments. In order to be more robust against outliers, we also analysed the median and four variants of truncated mean which can be seen as intermediate aggregation methods between mean and median.

The larger the distance to an object, the more noise can be observed in depth images. This makes approximation of the facial shape more difficult leading to lower quality of the Kinect's models. We therefore expect the measures for the proposed geometric features to be particularly noisy for records that were taken far away from the sensor. As these measures might have a negative impact on the quality of the aggregated features, we consider only a subset of the closest  $N$  records during our experiments.

In order to select a good classifier for our system, we use Rapid Miner [6] to evaluate the quality of our feature set. We tested two classifiers in our experiments, k-Nearest Neighbour (k-NN) [5] and Linear Discrimination Analysis (LDA)

and report a recognition rate of up to 88% for k-NN and up to 89% for LDA on a data set with 37 individuals.

These recognition accuracies are a significant improvement over our previous system while the aggregated features are still as comprehensible as the previously used raw features. Most importantly, the aggregation of feature values was shown to improve the recognition accuracy significantly. In order to be used in an access control system, we aim to further increase our system's performance.

### *C. Third Prototype*

As based on the Kinect's face and skeleton model, the first two versions of our prototype are not easily adoptable to the use of other sensor devices. Moreover, the Kinect's face and skeleton model have high demands with regard to hardware. This might be a problem once the system is in use on site where such machines are not available or increase the costs dramatically. Hence, we wanted to become independent of the Kinect SDK's advanced capabilities while we still use the sensor and its low-level API. The subset of the provided functionality that we use in the third version of the system is available for many other low-cost sensors. We acquire colour, infrared and depth frames as well as a mapping between these data sources. This functionality is also offered in OpenNI [7] for a variety of different sensors.

Although the recognition accuracies using aggregated geometric features are a significant improvement over the first version of our prototype they are not yet sufficient for reliable access control. However, they have shown that geometric features can contribute to reliable recognition in a comprehensible manner. To explore additional features and to improve the system's accuracy, we currently focus on colour, infrared and depth data directly which were not used in the previous systems. We aim to extract comprehensible features from these images as an intermediate representation. These features can again be geometric features as the distances between certain feature points but they are not restricted in this way and more importantly, no longer based on the Kinect's models.

As a basis for feature extraction we decided to generate frontal view images of detected faces. When a person approaches the sensor, his or her head and face are detected. We also estimate the person's head pose meaning that the exact position and orientation of a person's face is known. As the depth frame provides spatial information, this allows to render the scene from a normalized position in front of a person's face. In this way we obtain depth images of detected faces that are aligned in a predefined position.

Given the mapping from depth frame to the colour frame respectively the infrared frame, it is further possible to use these as a texture. Hence, we are able to render frontal views of a detected face using either the colour frame or the infrared frame as a texture. Three different kinds of frontal views of a person's face can be computed in this way. As the system is currently under development, we want to focus on this part of the data processing pipeline in what follows. These first steps as a part of the data processing pipeline are sufficient

to point out the idea of how such a system can be modelled using existing modelling frameworks.

### III. MODELLING FRAMEWORKS AND TOOLS

All three prototypes implement a data processing pipeline that transforms the acquired sensor data into a compact and characteristic feature representation through a sequence of small data transformations. Every single transformation takes one or more of the previously computed representations as its input and computes a new representation from them. As only the final outcome is of any interest while intermediate representations are solely used for the computation of the final outcome, all of our prototypes can be modelled intuitively by focusing on the data flow only. In fact, we will show that the control flow can be derived from the data flow in our example.

In the example presented in this paper, we are focusing on the newest version of our prototype. As the final recognition is not implemented to date, we will focus on the generation of frontal view images of peoples' faces which will be part of the final data processing pipeline. These frontal view images can be obtained from colour, infrared and depth data by rendering the acquired data from a changed viewport. In order to do so the face position and its orientation have to be estimated precisely. Our newest prototype approaches this task in four steps based on a single depth frame.

We show two example models of our prototype using two different modelling tools that were generated using the modelling framework Cinco. In what follows, we will first introduce the reader to the modelling framework Cinco and to Dime, the most complex Cinco-Product to date. We will further show a small custom Cinco-Product that models the data flow only and is tailored to the needs of our prototypes' models.

#### A. Cinco

Cinco is a meta modelling framework for graphical Domain Specific Languages that is developed at TU Dortmund University since 2014 [8] [9] [10]. It is based on the popular Eclipse Open-Source IDE and allows for the generation of Cinco-Products that are themselves based on the Eclipse IDE. Graphical Domain Specific Languages in Cinco are based on the concept of directed graphs meaning that a predefined set of custom nodes and edges is defined for a particular Cinco-Product. The meta modelling framework allows to define the appearance for each kind of node and edge and allows to constrain their connectivity. In this way it is possible to allow certain edges to connect only very particular kinds of nodes, but many other ways of constraining the graphical language are possible.

To enable rich features in Cinco-Products, the framework implements the concepts of hooks which allows to programmatically adjust the graph in case of a particular event. Such an event can be that a node was moved on the canvas or that it was removed from it. In particular, this allows to implement custom spatial arrangement of multiple nodes relative to one another but many other applications are possible.

As a meta modelling framework Cinco is used to generate modelling tools that are referred to as Cinco-Products. Due to the only assumption that a graphical Domain Specific Language is a directed graph, Cinco is very flexible and allows to generate modelling tools for a wide range of applications. Cinco itself does not associate any semantics with the graphical language but allows for the generation of an API that can be used to generate code from the graph models or to interpret them otherwise. Particularly interesting for our example models, edges can be used to represent both, control flow and data flow.

#### B. Dime

As the most complex Cinco-Product to date, Dime is the prime example of the Cinco's flexibility. As a Cinco-Product, Dime defines a set of nodes and edges, their appearance and also constraints the way they can be connected. While nodes represent situations during model's execution, edges are used to model both, control flow and data flow.

The most important nodes are the so called Service Independent Building Blocks (SIB) which represent executable code in the model. Every SIB has a list of input ports similar to function or method parameters in other programming languages. The functionality represented by the SIB relies only on the data provided by means of these input ports. The execution of a SIB can result in different cases which are modelled using the concept of branches. Every SIB must have one or more branches as its successors, each representing one case. Depending on the outcome of the execution of the SIB, one branch is chosen that determines the SIB that is to be executed next. In this way branches are used to model the control flow of the system. In addition, the selection of a particular branch, any other outcome of a SIB will be represented as variable. In Dime the set of computed variables can be defined for every branch separately. This is often appropriate as there will be no computation result in some error cases or different results can be computed in different cases. Dime represents the outcome in terms of data by output ports that are associated with the branch nodes. Figure 2 shows a small example of a Dime model.

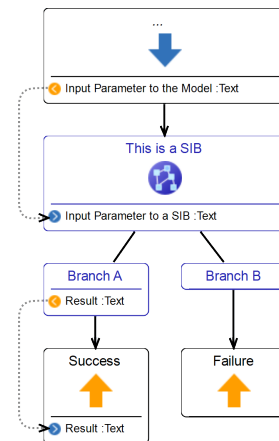


Fig. 2. Minimal example of a Dime model.

model with one SIB that has two branches only one of which has an output port.

As Dime allows the user to model control flow and data flow, it has to provide at least two different kinds of edges. In fact, there are many more kinds of edges but for the sake of simplicity, we want to focus on data flow and control flow. The control flow starts at the start SIB which is represented by a blue arrow. To make the entry point unique, there can only exist one start SIB in every model. Together with the end SIBs they are the only SIBs that have no branches. During execution the start SIB will do nothing as it is solely used to represent the start of the control flow and potential input ports. The end SIBs are used to represent different cases as an outcome of the model's execution and their associated output ports. As such they serve a similar purpose as branches on the level of the entire model. In fact, this is how Dime allows users to model in a hierarchical manner, meaning that the whole model can be used as a SIB in other models. In order to define the control flow from the start SIB to one of the possible end SIBs, the user has to define the control flow. This is done by connecting branches as the outcome of SIBs to exactly one other SIB. Depending on the outcome of the execution of a SIB, this allows to define the successor separately for every case. The control flow must be defined for every possible branch to make the model valid.

When the control flow reaches a SIB, all of its input ports must be available. The required data can be provided by the initial input parameters on the level of the entire model or it can be provided as the outcome of a previously executed SIB. In any case, the variable to be used as an input must be defined using data flow edges. These edges are dashed and connect exactly one output port of a branch to one input port of a SIB. Moreover, the start SIB's ports can be used as output ports and the end SIBs' ports can be used as input ports. It is the user's responsibility to define the data flow and the control flow in such a way that required input data is available when a SIB is reached. Hence, the data flow imposes constraints on the control flow and vice versa which can be exploited in the example that we present in this paper.

As Dime is a Cinco-Product and defines a set of nodes and edges, with clear interpretation, Dime is no longer as flexible as the use of Cinco for a tailored Cinco-Product. However, by modelling both, control flow and data flow its graphical models can express similar things as many programming languages in an intuitive fashion. Dime is still flexible in the sense that SIBs can have arbitrary functionality.

### C. Custom Cinco-Product

Although Dime is suitable for many applications as it allows to model both, data flow and control flow, there are applications that can be modelled more intuitively in other ways. In our example, we are only interested in the final outcome of the computation, respectively the final data representation. As every data transformation depends on one or more data representations, an order of all data transformations is implicitly defined by the data flow. Hence, this example

allows for modelling the data flow only while the control flow can be derived automatically.

For our second example we have therefore created our own Cinco-Product that allows for modelling the data flow only. There are three kinds of nodes, namely input representations (blue), output representations (green) and intermediate representations (white) and only one kind of edges to model data flow. All of these nodes represent a form of data that will be computed during the execution of the model if necessary. Note, that intermediate representations also represent a data transformation as they are computed from one or more other representations.

## IV. EXAMPLE MODELS OF OUR PROTOTYPE

All of our prototypes were developed in a way that allows to easily model their data processing pipeline using either Dime or a custom Cinco-Product. The recognition algorithm can be clearly separated into a sequence of data transformations as will be shown by means of the following two example models. We present example models for both of the modelling tools, Dime and our own Cinco-Product.

### A. Dime Model Example

As Dime allows for modelling control flow and data flow, the data processing pipeline can be easily modelled in Dime. Each of the data transformations as part of the data processing pipeline can be represented as a SIB. The required input representations are inputs to the SIBs and will therefore be connected to the SIBs' input ports. In our example we want to focus on the data flow and we want to show that the control flow can be automatically derived from it. For the sake of simplicity this example is therefore limited to a single branch per SIB. When the model is used to generate code in future versions of our system, more than one branch will be necessary to handle exceptions in any of the data transformation steps. For instance, there might be no person visible in an acquired frame and hence no meaningful head pose can be computed.

Ignoring these exceptions in the current version of the model, every SIB has exactly one branch which is the success branch. The success branch is necessary to provide outputs of the computation, namely a new data representation. In the example new frames are acquired from each of the three sources as a very first step. The first three SIBs fulfil this purpose and provide colour, infrared and depth frames as output ports of their success branches. While colour and infrared frames are solely used as a texture for the generation of the frontal view images, the depth frame plays an important role. As it provides information about the facial shape it can be used to approximate the head pose. In the newest version of our prototype, this problem is approached as a sequence of four refinements as shown in the second row of the model in Figure 3. First, the head position is roughly approximated from the raw depth image as given to the first SIB's input port. The success branch provides the head position approximation which serves as an additional input for the more precise head position computation. Both, the raw depth frame and

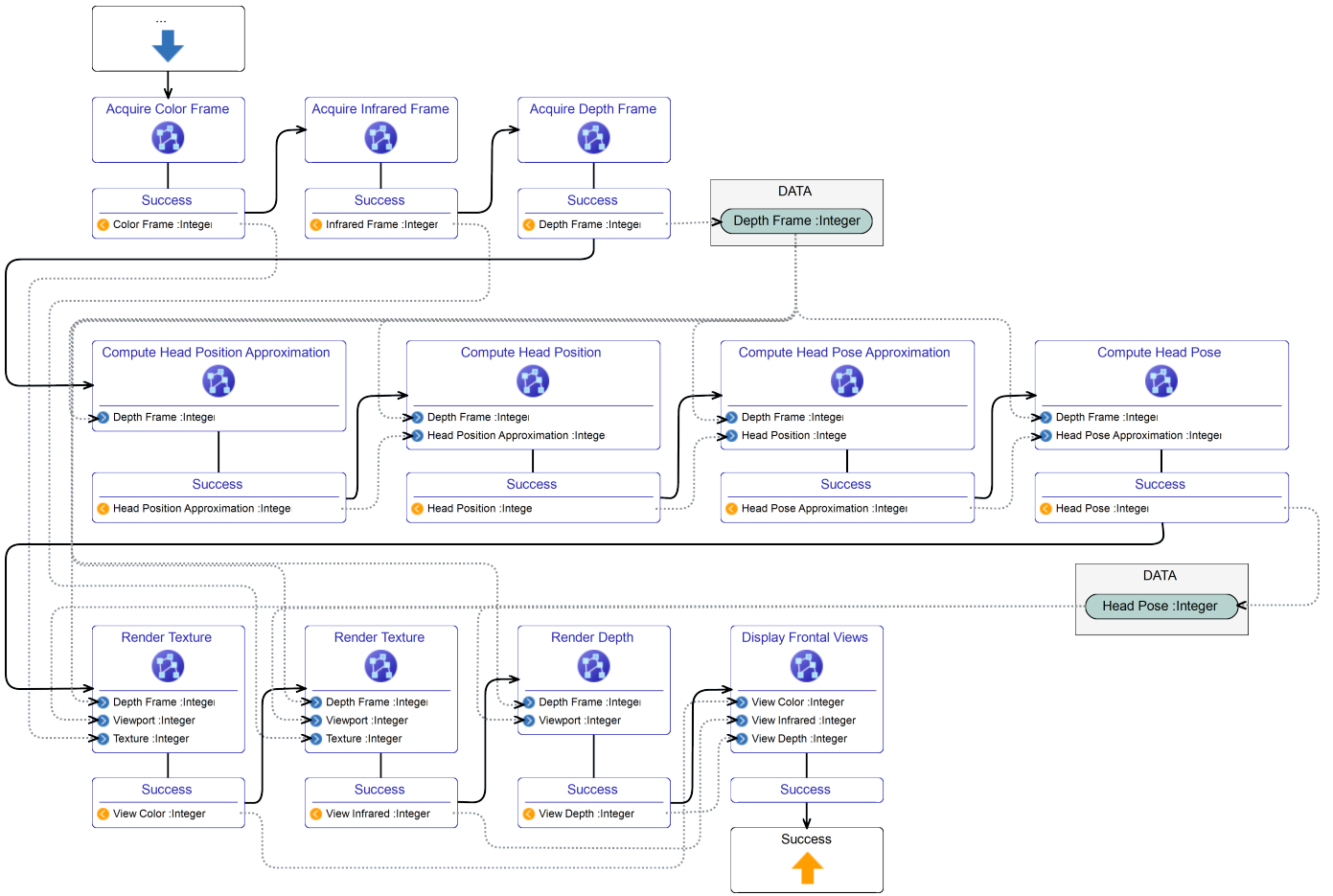


Fig. 3. Dime model for the generation of frontal view images from colour, infrared and depth data.

the head position approximation are connected to the head position computation SIB's input ports. In a similar fashion, the head pose is computed in two consecutive steps. Finally, a precise estimate of the head pose is available which allows for rendering of frontal views.

Given the head pose, the first three SIBs in the third row of our model in Figure 3 render the frontal view images. As input parameters, all of them expect the raw depth image as acquired from the sensor which provides spatial information and also the precise head pose estimate which defines the viewport from which the scene is to be rendered. In addition, colour and infrared images are used as a texture leading to three different frontal view images of the detected face. For demonstration purposes the last SIB takes all of these images as inputs and displays them.

Note, that some data representations such as the depth image and the head pose are used more than once. In Dime this requires the use of a data context that holds variables and allows for them to be used multiple times. In general, this is necessary as SIBs can change the value of their input parameters. However, in our example all of the input variables are only read which allows for using them multiple times in an arbitrary order.

Our goal is to define reusable components from the data processing pipeline of our final version of the admission control system and to provide them as SIBs in Dime. Not only would this allow to modify the system in a very intuitive way and would allow non-programmers to adjust the system at any time, but also would this allow for building similar systems from the existing components in a very easy way. Especially people with little to no programming skills would be enabled to create advanced systems that detect people, their head poses and many other things depending on the capabilities of the palate of provided SIBs.

### B. Custom Cinco-Product Model Example

As only the displayed image the displayed image in this example as the final outcome of our model's execution is of any interest, the order of execution is irrelevant as long as required input representations are available for every data transformation in the data processing pipeline. In order to exploit this property, we use our custom Cinco-Product to model the data flow of the pipeline only. Every node represents data while intermediate data representations (white) implicitly represent data transformations that define how the new data representation can be obtained from others. In the example,

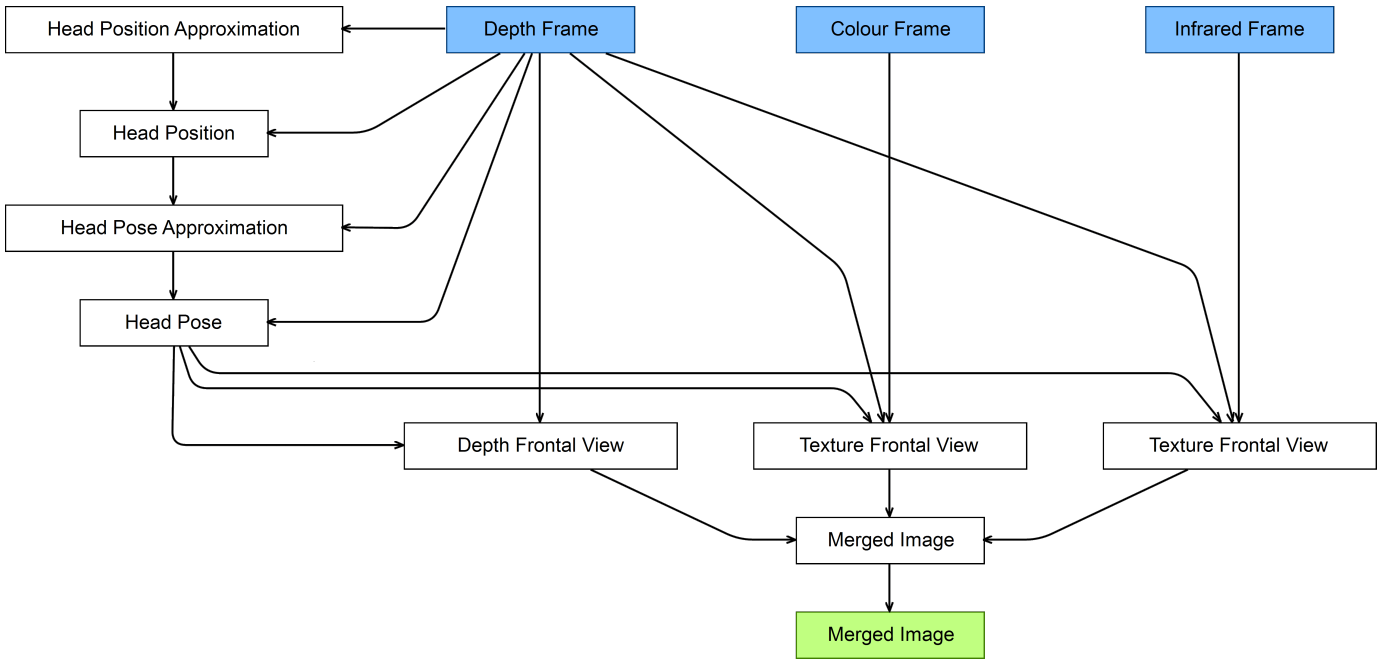


Fig. 4. Cinco-Product model for the generation of frontal view images from colour, infrared and depth data.

raw sensor data is given as input representations (blue). In particular, these are colour, infrared and depth frames that were acquired using the Kinect sensor. Per execution of the pipeline one frame from each source is available and all of them can be used to obtain the final data representation. As the newest version of our prototype does not implement the final recognition of an individual yet, we limit the example to the generation of a collage of frontal view images from all three sources. The frontal views will later be used to extract facial features for the recognition of individuals. In order to render frontal view images of faces, the head pose must be known which defines the viewport from which the scene has to be rendered.

As the prototype is the same one that was used for the Dime model example, the computation of the head pose is again approached in four steps. First, the head position is approximated in the raw depth frame. The data representation *Head Position Approximation* implicitly represents the data transformation from a raw depth frame to an approximation of a person's head position. The new data represents only the approximation of the head position and no longer the depth frame. It is therefore a significantly smaller data representation than the *Depth Frame* which was provided as an input representation. In a second step, the *Head Position* is computed more precisely from the raw *Depth Frame* and from the *Head Position Approximation*. The new data representation therefore depends on two others which have to be available before the *Head Position* can be computed. Hence, the data flow as defined in the model in Figure 4 imposes constraints on the order of data transformations.

Note, that the model in Figure 4 does not define the control flow but only the data flow. As the data flow imposes

constraints on the order in which data representations must be available, a possible control flow can be deduced automatically from the topological order of the graph. More precisely every input representation must be available before a new data transformation can be applied. In our example, this means that *Head Position Approximation*, *Head Position*, *Head Pose Approximation* and *Head Pose* must be computed in exactly this order before any of the other data representations can be derived. For the generation of the separate frontal view images, the order can be arbitrary as they do not depend on one another but only on input representations and the *Head Pose*. Finally, the *Merged Image* must be computed at the very end. This is also defined as the final output representation (green) of our model. Any case in which the order of computation is irrelevant allows for parallelism. In our example, the generation of the separate frontal view images can in fact be performed in parallel once their input representations are available.

As an alternative to SIBs in Dime our custom Cinco-Product has strong focus on data flow. While this simplifies modelling of a data processing pipeline, there is no easy way of modelling side effects, defining the order of execution etc. This Cinco-Product is nevertheless useful for data oriented applications such as processing pipelines in computer vision systems similar to the one in our example.

## V. CONCLUSION

In this paper, we presented three generations of prototypes for a contactless admission control system with high potential to be modelled with available modelling frameworks and tools.

We presented the three versions of our prototype and their commonalities and differences. In particular, we focused on the data processing pipeline which all prototypes implement

in a similar fashion. This part of the system can be modelled intuitively with modelling tools such as Dime or our custom Cinco-Product.

In order to show our prototypes' potential to be modelled, we introduced the reader to Cinco, Dime and our custom Cinco-Product. Focusing on the first part the newest processing pipeline, we show examples of models in both tools. We discussed the possibility to derive the control flow automatically from a specification of the data flow in the data processing pipeline.

We continue to develop the most recent version of our prototype in a way that maintains its high potential to modelled visually. This will enable us to define reusable components from the data processing pipeline of our final admission control system and to provide them as SIBs in Dime. Moreover, we aim to model the system using a similar Cinco-Product to the one that we presented in this paper. Not only would this allow to modify the system in a very intuitive way and would allow non-programmers to adjust the system at a later stage, but also would this allow for building similar systems from the existing components in a very easy way. Especially people with little to no programming skills would be enabled to create advanced systems that detect people, their head poses and many other things. Once a set of powerful SIBs, respectively data transformations is developed for computer vision related applications, it can be extended continuously leading to a rich palate of SIBs. Depending on the extend of this palate, this would allow for modelling a wide range of computer vision related applications. In the long term, such a palate could also be extended to an even broader range of systems that implement any kind of a data processing pipeline.

#### ACKNOWLEDGMENT

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre ([www.lero.ie](http://www.lero.ie)).

#### REFERENCES

- [1] G. S. Gagandeep Kaur and V. Kumar, "A review on biometric recognition," *International Journal of Bio-Science and Bio-Technology*, vol. 6, no. 4, pp. 69–76, 2014.
- [2] Z. Zhang, "Microsoft kinect sensor and its effect," *IEEE MultiMedia*, vol. 19, no. 2, pp. 4–10, 2012.
- [3] F. Gossen, "Bayesian recognition of human identities from continuous visual features for safe and secure access in healthcare environments," in *Design Technology of Integrated Systems in Nanoscale Era (DTIS), 2015 10th International Conference on*, 2015.
- [4] I. Marqués and M. Graña, *Computational Intelligence in Security for Information Systems 2010: Proceedings of the 3rd International Conference on Computational Intelligence in Security for Information Systems (CISIS'10)*, ch. Face Processing for Security: A Short Review, pp. 89–96. 2010.
- [5] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.
- [6] "Rapid miner." <https://rapidminer.com/>. Accessed: 2016-02-02.
- [7] "Openni 2 sdk." <http://structure.io/openni>. Accessed: 2016-04-01.
- [8] S. Naujokat, L.-M. Traonouez, M. Isberner, B. Steffen, and A. Legay, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, ch. Domain-Specific Code Generator Modeling: A Case Study for Multi-faceted Concurrent Systems, pp. 481–498. Springer Berlin Heidelberg, 2014.
- [9] S. Naujokat, M. Lybecait, B. Steffen, D. Kopetzki, and T. Margaria, "Full generation of domain-specific graphical modeling tools: a meta modeling approach." under submission, 2015.
- [10] "Cinco scce meta tooling framework." <http://cinco.scce.info/>. Accessed: 2016-04-01.



# System for Deep Web Users Deanonimization

Aleksandr Lazarenko

Software Engineering School

National Research University Higher School of Economics

Moscow, Russia

avlazarenko@edu.hse.ru

Scientific Advisor: Prof. Sergey Avdoshin

Software Engineering School

National Research University Higher School of Economics

Moscow, Russia

savdoshin@hse.ru

**Abstract**— Privacy enhancing technologies (PETs) are ubiquitous nowadays. They are beneficial for a wide range of users. However, PETs are not always used for legal activity. The present paper is focused on Tor users deanonymization using out-of-the box technologies and a basic machine learning algorithm. The aim of the work is to show that it is possible to deanonymize a small fraction of users without having a lot of resources and state-of-the-art machine learning techniques. The deanonymization is a very important task from the point of view of national security. To address this issue, we are using a website fingerprinting attack.

**Keywords**—Tor, deanonymization, website fingerprinting, traffic analysis, anonymous network, deep web

## I. INTRODUCTION

Internet privacy is considered to be an integral part of freedom of speech. A lot of people are concerned about their anonymity in public and therefore there is a growing need for privacy enhancing technologies.

The Deep Web is the layer of the Internet which can not be accessed by traditional search engines, so the content in this layer is not indexed. The typical website in the deep web is static with potentially no links to outer resources. That is why it is very hard to measure the real size of the deep web.

In the modern world, there are a lot of networks and technologies which grant access to the deep web resources, for example, Tor, I2P, Freenet, etc. Each of these instruments hides users traffic from adversaries, thus making the deanonymization a hard thing to do. A detailed overview of such technologies can be accessed in paper [1].

Nowadays, the largest and most widely used system is Tor [2]. Our research is focused on Tor users deanonymization, because of its popularity and prevalence.

## II. TOR BACKGROUND

Tor is the largest active anonymous network in the world. There are more than two million users per month and the number of relays is close to 7000 [3]. Tor is a distributed overlay network which consists of volunteer servers. Every user in the world can provide Tor with computational resources needed for traffic retranslation over the network.

Despite of being a great privacy enhancing technology for law-abiding citizens, Tor is an essential tool in criminal

society. Terrorists, drug and arm dealers in line with other offenders use Tor for their criminal activities. Thus, the solution of the deanonymization problem is very important for government special services [4]. For example, Russian Ministry of Internal Affairs (MIA) has recently announced a bidding for Tor deanonymization system [5].

The next key component of Tor is Hidden Services (HS). Tor HS provides users with anonymous servers to host their websites or any other applications. HS are accessed via special pseudo-domains «.onion», where The Deep Web is located. From the user's point of view, accessing a particular hidden service is as easy as visiting a normal website.

In order to establish a connection with Tor network, the user must have pre-installed software (Tor client). The easiest way is to install TorBrowser, which is a customized version of Mozilla Firefox with built-in Tor software. To initiate the connection, a Tor client obtains a list of Tor nodes from a directory server. Then, the client builds a circuit of encrypted connections through relays in the network. The circuit is extended hop by hop and each relay on the path knows only which relay gives data and which relay it is giving data to. There is no particular relay in the circuit (see Fig. 1) which knows the complete users path through the network.

A Layered encryption is used along the path. The most interesting relays for a potential attacker are entry and exit relays. Every piece of information in the network is transferred in the Tor cells, which have an equal size. An Entry relay (also called the guard) knows the IP address of the user and Exit relay knows the destination resource. Traffic interception in the middle would not give any advantage to the attacker because everything is encrypted and secure.

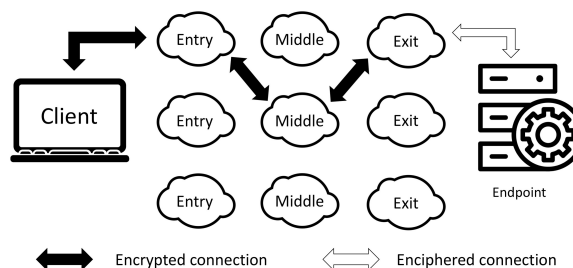


Fig. 1. Tor circuit example

### III. DEANOMIZATION TECHNIQUES

There is a wide range of deanonymization methods (attacks). Some of them are passive: an adversary only observes traffic, without any trials to modify it somehow. Contrariwise, some of them are active: an attacker modifies traffic (causing delays, insert patterns, etc.). We proposed a classification of attacks, where the main principle is the amount of resources needed by an attacker to perform the deanonymization (see Table 1).

TABLE I. CLASSIFICATION OF DEANOMIZATION TECHNIQUES

#	Resources	Attacks
1	Corrupted entry guard	<ul style="list-style-type: none"> <li>Website fingerprinting attack</li> </ul>
2	Corrupted entry and exit nodes	<ul style="list-style-type: none"> <li>Traffic analysis</li> <li>Timing attack</li> <li>Circuit fingerprinting attack</li> <li>Tagging attack</li> </ul>
3	Corrupted exit node	<ul style="list-style-type: none"> <li>Sniffing of intercepted traffic</li> </ul>
4	Corrupted entry and exit nodes, external server	<ul style="list-style-type: none"> <li>Browser based timing attack with JavaScript injection</li> <li>Browser based traffic analysis attack with JavaScript injection</li> </ul>
5	Autonomous system	<ul style="list-style-type: none"> <li>BGP hijacking</li> <li>BGP interception</li> <li>RAPTOR attack</li> </ul>
6	Big number of various corrupted nodes	<ul style="list-style-type: none"> <li>Packet spinning attack</li> <li>CellFlood DoS attack</li> <li>Other DoS and DDoS attacks</li> </ul>

More information about attacks mentioned in Table 1 can be accessed in paper [6]. We are focused on the resource-effective attack (WF), which only requires an attacker to control an entry relay of the user. The relay which is fully controlled by an attacker is called a *corrupted relay*.

### IV. WEBSITE FINGERPRINTING ATTACK

#### A. Website Fingerprinting Attack Overview

A website fingerprinting attack (WF) is an attack designed for a local passive eavesdropper to determine the client's endpoint using features from packet sequences. Generally speaking, WF breaks privacy which is achieved by the proxy, VPN or Tor. This is an application of various machine learning techniques in the field of privacy.

The first appearance of the WF was discussed in paper [7]. This attack has been widely discussed in the researchers' community because it has proven its effectiveness against various privacy enhancing technologies, such as Tor, SSL and VPN.

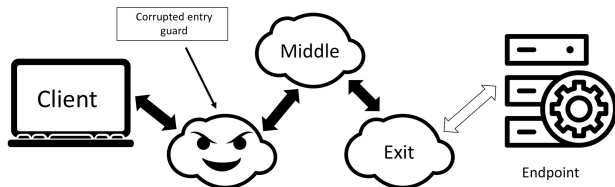


Fig. 2. Configuration of Tor circuit suitable for the WF attack

To perform a WF, an eavesdropper has to simulate users' behavior in the network, using the same conditions as the victim. In case of Tor, an attacker must have a corrupted entry relay (see Fig.2) that will be used for collecting data. The Attacker visits each site from the list and stores all packet sequences related to the request. Then he uses the traffic for training a classifier in a supervised way. The machine learning problem could be stated as a binary classification problem or multilabel classification problem. In the first case, classifier is trained to answer the question: «If the user visits a site from our list?». The second option is about guessing a particular website that the user visits.

#### B. The Oracle Problem

Since WF works with packet sequences, determining the sequences related to the webpage is quite a difficult task. This issue is known as the Oracle problem. Researchers make two major assumptions, which simplify WF a lot: 1) an attacker has such an oracle at his disposal, 2) the victim loads pages one-by-one in a single tab. The Oracle helps to find precise subsequence of packets from overall captured traffic. Any excess packet sequence sent to classifier can significantly reduce its' accuracy. That is why splitting the whole sequence is crucially important. Another reason is the user's web-browsing behavior. The majority of people uses multi-tab browsing instead of loading a page in a single tab, working with it and loading another one. This behavior makes WF difficult in real life.

An oracle problem for packet sequences has not been solved yet, but Wang proposed a solution for Tor, which can work with a single tab. He proposed a three-step process of determining correct split in case of single tab browsing between two pages. Wang used Tor cells instead of packets. The first step is making a time based split. The Attacker splits sequences if the time gap between two adjacent cells is greater than some constant, then the sequence is split there into two subsequences. If the time gap is too small, classification-based splitting is typically used. Wang uses machine learning techniques that decide where to split and whether or not to split. After splitting, the result is ready for further classification. This method achieves quite good accuracy. However, the proposed solution doesn't work with multi-tab browsing and raw packet sequences, narrowing the range of real implementations. Study [8] proposed a time-based way to split traffic traces when the user utilizes 2 open tabs. They classify the first page with 75.9% and second with 40.5% accuracy.

#### C. Real World Scenario

Overall, the applicability of WF in the real world scenario is still questionable. Users may visit hundreds of thousands of webpages every day. So, can the attacker successfully apply WF in reality? Panchenko et al. [9] checked the attack with a really huge dataset and their approach outperformed the previous state of the art attack proposed by Wang. To conclude, WF attacks are still a serious threat to anonymous communication systems.

The aim of the current work is to show that an attacker can build a deanonymization system, applying learning libraries on

most popular programming languages, which will be able to deanonymize a group of users that try to access the deep web content.

## V. DEANONIMIZATION SYSTEM SCHEME

For the sake of simplicity, we will use as much preconfigured software as possible. In order to deal with deanonymization problem, our system must have two modules. The first module is for mining Tor data which will be used for collecting traffic traces. The second is for applying machine learning techniques.

### A. Data Mining

The data mining module is using various software, which can be easily installed on Mac OS or any Linux distributive. Since the packet traces can be collected on the relay side, or on the client side (the difference is only in the source/destination pair), we can use data mining module on local machine or on the remote server. We will use local machine for data mining (see Fig. 3). Simple data transformation can be applied for packet traces, to look exactly like those collected on the relay.

The following software must be installed on the machine:

- Tor - free software for enabling anonymous communication.
- Torsocks - free software that allows using any kind of application via the Tor network.
- Wget – a program which retrieves content from the web server, supports downloading via http, https, ftp.
- Tshark – a free and open packet analyzer. It is used for network troubleshooting, analysis, etc.
- Mozilla Firefox or Tor Browser - an open web-browser. In case of Mozilla Firefox, you will need to configure it for using Tor manually.

Nevertheless, any program can be replaced by the specific library. The simplest solution is to use the proposed software. We must have full control over Tor circuits construal to use our own relay. For this purpose, we will use Stem Python library, which is freely accessible on the web. Stem is a Python controller library for Tor.

We use Stem to create Tor circuits through our corrupted entry guard. Without this action, the accuracy of the classifier might become worse, because of different Tor versions on the relays and other reasons. Another option is to modify Tor configuration for using specified entry guards. It is very important to use the same entry guard which will be used in production.

Tshark is used as the main packet capturing tool. We also use Tshark for extracting TLS records from data. Tshark can be substituted with any library, which supports capturing of TCP packets.

After that, the attacker has to automate the data gathering process. There are two ways to do it, namely, using wget via torsocks, or Mozilla Firefox. In case of wget, an attacker just launches page downloading from the command line, but

Mozilla Firefox requires more work. The automation of Mozilla can be done in two ways. The first option is to launch it from the command line and wait while the page is uploading, another one is to use Selenium Webdriver to automate the process.

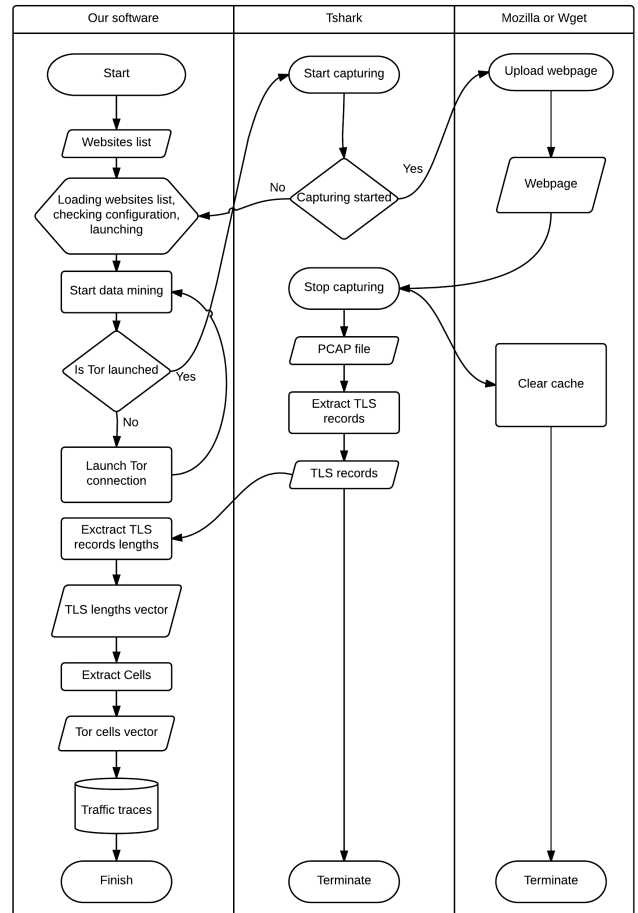


Fig. 3. Data mining process

### B. Feature Extraction

We can extract features of the traffic at three different levels (see Fig.3) – they are Tor cells, TLS, TCP. On the application level, Tor retranslates data in the fixed size packets, called cell. All cells have an equal length, which is 512 bytes, and travel throughout the network in TLS records. It is noteworthy that several cells can be packed in a single TLS record. The last level is transport level: TLS record is then fragmented into several TCP packets. TCP packets size is limited by the MTU. Furthermore, several TLS records can be packed into a single TCP packet. However, it is questionable which level is the most informative from the website fingerprinting attack perspective. The majority of researchers assume that the most informative level is the cells level.

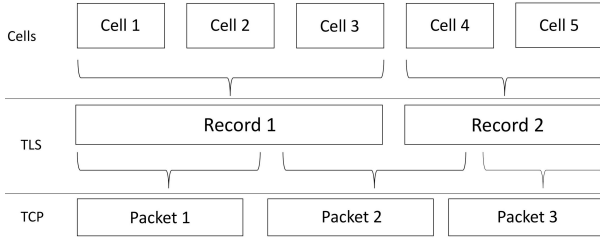


Fig. 4. Information extraction levels

Firstly, the cell traces extraction should be performed in the following way: an attacker must extract TLS records from TCP packets. It could be achieved with the tshark software.

Here the file\_name should be substituted by the .pcap file with TCP packets, whereas output\_file is the desired output file with textual representation of TLS records. Hence, a simple regular expression can then be used for length extraction. Once the number is an extended extraction, an attacker should then multiply it by -1 if it is outgoing.

The resulting array of TLS records lengths should then be transformed into Tor cells. An attacker should divide each number by 512 and append to the cells vector as many -1 or 1 as the number of integers in the result of division. For example, if the length of TLS record is equal to 2048, the resulted cells vector would be [1,1,1,1].

After cell traces extraction, we will have the following representation of the data [-1,1,1,1,-1,...]. Such arrays are then used as features, subsequently, the actual webpages are then used as labels. However, such arrays have different lengths. Hence, as we are trying to simplify the process, we will append zeros to the end of input vectors because the majority of machine learning algorithms requires the input vectors to have equal lengths. By means of such operation, we will equalize the length of the cell vectors.

### C. Machine Learning Module

For machine learning purposes we will use *sklearn* Python library, which is the most popular Python library for machine learning. The trained model will be used for classification of new traffic samples.

This module works in a straightforward way. An attacker must train the model using collected cells and then use it as a ready model.

## VI. EXPERIMENTAL SETUP

We have implemented such a scheme using Java programming language and Python (Fig. 5). The aim of our experiment is to show that we can deanonymize a small fraction of users in the real world even if we don't use cutting-edge deanonymization techniques.

### A. Experimental Environment

Consider the following situation: the group of terrorists is trying to gain access to illegal content from a small room in the dormitory. The list of resources was provided by the Group-IB cybersecurity company. In our experiment there were three

users playing the role of terrorists. Each of them visited the resources from the list according to the following rules: only single tab browsing and the time spent to read the webpage is at least 5 seconds. According to the research, this is realistic [10]. Such rules allow us to simplify the process of splitting packet sequences and extracting traces.

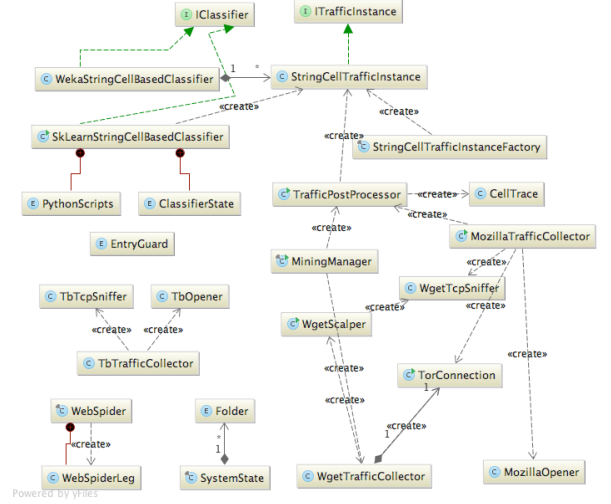


Fig. 5. UML class diagram of traffic collection module

### B. Data Gathering

Before trying to deanonymize users, we made a preparation step and collected 80 traffic instances from our list of resources. Such a low number of traffic instances is sufficient because bigger datasets are not increasing accuracy of classifier on the same number of websites. We have studied 7 resources related to drugs, weapons and extremism issues.

Our users repeated the process of reading and uploading a webpage for 5 times for each webpage from the list. After that, we downloaded collected packet sequences and made the data preprocessing step. We used time-based splitting as was done by Wang [11]. After this step, our data became ready by the classifier.

### C. Machine Learning Model

Support vector machines (SVM) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. An SVM model represents the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted as belonging to a category based on which side of the gap they fall to.

We used the NuSVC machine learning algorithm with default hyperparameters from the *sklearn* library. NuSVC is Nu-Support vector classification based on the support vector machines. This algorithm uses a parameter to control the number of support vectors, where the parameter is an upper

bound of the fraction of training errors and lower bound of the fraction of support vectors.

## VII. EVALUATION

### A. Evaluation metrics

- True positives (tp) - equal with hit.
- False positives (fp) - equal with correct rejection.
- False negatives (fn) - type 2 error.
- Precision - the ratio  $tp / (tp + fp)$ ; there is an intuitive ability of the classifier to avoid labelling a negative sample with the positive label.
- Recall - is the ratio  $tp / (tp + fn)$ ; there is an intuitive ability of the classifier to find all the positive samples (the best is 1, the worst is 0).
- F1-score - is a weighted average of the precision and recall (its best value is 1, worst 0) =  $2 * (precision * recall) / (precision + recall)$ .
- Score – the subset accuracy returns in a multilabel classification. If the entire set of predicted labels for a sample strictly matches with the true set of labels, then the subset accuracy is 1.0, otherwise it is 0.0.

### B. Experimental results

We have performed the classifier evaluation using a built-in *sklearn* function. For ethical reasons documented in Tor ethical research [12], we've anonymized the websites used for the experiment.

Our simple model achieved the following results:

TABLE II. CLASSIFIER EVALUATION

Website	Precision	Recall	F1-score
Site 1	1.00	1.00	1.00
Site 2	0.80	0.80	0.80
Site 3	0.80	0.80	0.80
Site 4	0.50	0.40	0.44
Site 5	1.00	1.00	1.00
Site 6	0.38	0.60	0.46
Site 7	0.67	0.40	0.50
Avg/total	0.73	0.71	0.72

Overall, the total score of the classifier = 0.714

These results are not outstanding in comparison with the state-of-the-art techniques, but they show that we can deanonymize users with the help of a relatively simple program and achieve sufficient accuracy.

## VIII. CONCLUSION

It was shown that an attacker without cutting-edge machine learning techniques can apply website fingerprinting. If the attacker has enough experience and technical competence, he will be able to build such a system and use it for the purpose of

deanonimization. Moreover, the proposed solution will work best if the attacker sniffs Wi-Fi or other local network because it is very easy for him to find tor related traffic and collect traces. In this case, the deanonymization is targeted and easily implemented.

## IX. FUTURE WORK

In our future work we are going to solve the oracle problem using the recurrent neural networks and test them in the field of website fingerprinting attacks. Next, we are going to build a cloud application as proposed in [13] using state-of-the-art techniques and results based on RNN research.

The main purpose of solving the oracle problem is to have a pretty accurate splitting algorithm, which will allow to use WF attacks even with the multi tab browsing.

## REFERENCES

- [1] S.M. Avdoshin, A.V. Lazarenko, "Technology of anonymous networks," Information Technologies, vol. 22, №4, pp. 284-291.
- [2] R. Dingledine, N. Mathewson, P. Syverson, "Tor: The Second-Generation Onion Router," in Proceedings of the 13th USENIX Security Symposium, August 2004, URL: <http://www.onion-router.net/Publications/tor-design.pdf> (accessed: 3.04.2016).
- [3] Relays and bridges in the network. Tor METRICS. URL: <https://metrics.torproject.org/networksize.html> (accessed: 15.02.2016).
- [4] The NSA's Beet Trying to Hack into Tor's Anonymous Internet For Years. Gizmodo [Official website]. URL: <http://gizmodo.com/the-nsa-been-trying-to-hack-into-tors-anonymous-inte-1441153819> (accessed: 28.09.2015)
- [5] Zakupka No0373100088714000008. State Procurements. URL: <http://zakupki.gov.ru/epz/order/notice/zkk44/view/common-info.html?regNumber=0373100088714000008> (accessed: 2.10.2015).
- [6] S.M. Avdoshin, A.V. Lazarenko, "Tor Users Deanonymization Methods," Information Technologies, vol. 22, №5, pp. 362-372.
- [7] X. Cai, X.C. Zhang, B. Joshy, R. Johnson, "Touching from a Distance: Website Fingerprinting Attacks and Defenses", URL: <http://www3.cs.stonybrook.edu/~xcai/fp.pdf> (accessed: 20.03.2016).
- [8] X.Gu, M.Yang, J.Luo, "A Novel Website Fingerprinting Attack Against Multi-Tab Browsing Behavior," in Computer Supported Cooperative Work in Design (CSDW), 2015, URL: [http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=7230964&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxppls%2Fabs\\_all.jsp%3Farnumber%3D7230964](http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=7230964&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxppls%2Fabs_all.jsp%3Farnumber%3D7230964) (accessed: 3.04.2016).
- [9] A. Panchenko, F. Lanze, A. Zinnden, M. Henze, J. Pannekamp, K. Wehrle, T. Engel, "Website Fingerprinting at Internet Scale," URL: <https://www.comsys.rwth-aachen.de/fileadmin/papers/2016/2016-panchenko-ndss-fingerprinting.pdf> (accessed: 1.04.2016).
- [10] T. Wang, "Website Fingerprinting: Attacks and Defenses", PhD Thesis, 2015, URL: [https://uwspace.uwaterloo.ca/bitstream/handle/10012/10123/Wang\\_Tao.pdf?sequence=3](https://uwspace.uwaterloo.ca/bitstream/handle/10012/10123/Wang_Tao.pdf?sequence=3) (accessed: 3.04.2016).
- [11] J.Nielsen, "How Long Do Users Stay on Web Pages," URL: <https://www.nngroup.com/articles/how-long-do-users-stay-on-web-pages/> (accessed: 20.03.2016).
- [12] "Ethical Tor Research: Guidelines," URL: <https://blog.torproject.org/blog/ethical-tor-research-guidelines> (accessed: 9.05.2016).
- [13] A.V. Lazarenko, "Structure of intellectual system of Tor Users Deanonymization," in Proceedings of Armenskogo Conference, 2016, URL: <https://www.hse.ru/data/2016/03/04/1125807985/MIEM-HSE-2016.pdf> (accessed: 3.04.2016).



# Model of security for object-oriented and object-attributed applications

Pavel P. Oleynik, Ph.D.

System Architect Software, Aston OJSC, Associate  
Professor, Shakhty Institute (branch) of Platov Southern  
Russian State Polytechnic University (NPI),  
Rostov-on-Don, Russia  
[xsl@list.ru](mailto:xsl@list.ru)

Sergey M. Salibekyan, Ph.D.,

Associate Professor; Institute of Electronics and  
Mathematics, Russia of Natal Research University "Higher  
School of Economics", (NRU HSE),  
Moscow, Russia  
[ssalibekyan@hse.ru](mailto:ssalibekyan@hse.ru)

**Abstract** The paper provides a survey of current approaches to the security organization and access control in various architectural applications. The paper presents the author's approach to the permission assignment in classes, attributes, and objects satisfying certain criteria. This is done by using the hierarchy of classes whose composition and structure are described in detail in the paper. Also the applications based on the security model already implemented by the authors are described. At the end of the paper, it is proposed approach to the security organization in object-attribute system

**Keywords:** *Security of information systems; Object-oriented applications; Object System Metamodel; Model of Permissions; object-attribute approach.*

## I. INTRODUCTION

At present, the greatest number of new applications is being developed by an object-oriented approach. This paradigm, based on the inheritance technology, allows one to reuse the previously developed elements implemented as classes. The result is the reduced development time and the costs of the whole information system. This is the key advantage when large software products are created. Such systems are typically multi-user systems. At the same time, each category of user needs is only a part of the available information, i.e. there is a problem of access control for multi-user applications. The paper presents a model of access control for object-oriented applications, which was developed by the authors and repeatedly used when developing large applications, and a model of access control in object-attribute computation system.

The paper is organized as follows. Section 1 provides a detailed survey of the papers devoted to similar topics. Section 2 describes the model of access control used by the author. Section 3 shows real examples of implementation of this model and the selected roles of users. Section 4 shows the approach to security in Object-attribute system. At the end of the paper, conclusions on this work and plans for the further study are given.

## II. A SURVEY OF THE AVAILABLE RESEARCH

Access permission is one of the main problems appearing after the development of the required functionality of the program. Therefore, there are a lot of researches representing different approaches to solving this problem. In [1], the authors propose an approach called business-oriented development (Business-Driven Development), in which the key role is given to the security configuration in the application. The authors use the Model-Driven Architecture (MDA) of architecture of the program. They introduce the concepts of business processes and models at the model level, and then determine the security policies and templates specifying certain rules for them. The present research describes principles of access permission assignment at the level of platform-independent models and the further transformation into platform-dependent models. As a result, the authors present a set of templates for access control providing that their configuration can be adjusted if necessary. This solution is tested using a service-oriented architecture (SOA). To improve the efficiency of the description of the software product life cycle and the corresponding access permissions, the authors propose to make several changes in the languages of software development, such as UML and BPEL. An advantage of the paper is the presence of a number of charts illustrating the proposed solution, as well as many code fragments represented as XML.

The research [2] is more practical and special. It describes a model of adaptive security for multi-agent information systems used by the authors in the medical information system called HealthAgents. The authors start from describing the classical model of access control based on Role-based access control (RBAC) and extend it to be used in multi-agent systems. In their research, the authors present a meta-model that allows one to manage access control by using the UML class diagram. To interact with the security role, the authors introduce the base class Subject attributed with different user permissions. The derived class represents users, organizations and agents. An analysis of research shows that the object-oriented approach for describing access rights is implemented. To describe the process of applying the security policies, the authors depict the Interaction Diagram and present, in the XML-code, an example of test description of access rights of certain users, stored in the system.

The research [3] presents the simulation of multi-level security, integrated within a service-oriented application. In a service-oriented architecture (SOA) that allows one to develop different Web applications, the security is critical. The security is provided by the Web service WS-Security controlled by SOAP messages. These messages may be attacked either by anonymous customers or by trusted clients. In addition, there are other possible types of attacks, for example, the so-called denial of service (DoS), which can exhaust the computer resources and make the Web service unavailable. The described security model consists of three levels. Attention is paid to each of the levels. The obtained multi-level security architecture is presented graphically, namely, various security domains, as well as the composition and structure of the software installed on each of them are depicted. After this, various types of possible attacks at each of the levels are discussed. They are described using the UML Class Diagram. This allows one to analyze the results obtained by the authors and then to design the desired security models based on the results.

The framework for describing the security model of service-oriented applications (SOA) is presented in [4]. The authors focusing on the process of modeling business processes use the BPEL notation. The security model is used with the model of business processes. The authors argue that the difference in approaches of a Business analyst and an Expert to solving the security problems leads to certain permission assignment that ultimately compromise the safety of user data. The authors developed several annotations that allow the security Experts to specify the security model. The proposed approach is demonstrated by an example of business processes of a service-oriented information system providing data about the progress of students. The paper describes a possible implementation of the framework, its basic modules and rules of interaction between the experts and the system.

The paper [5] presents model-oriented templates (patterns) of application security obtained by the authors by an analysis of phases of the application development. The authors examine the applications working in Internet. The templates contain descriptions of solutions to common security problems. The selection of an appropriate pattern depends not only on the situation but on other templates applied earlier, i.e. the dependence between the patterns is taken into account. The authors present an analysis of such dependencies for the first time. The technology of changes of General security templates is proposed on the basis of a rule transformation model based on previously used patterns. This allows one to avoid inappropriate application of the security templates. The authors identify two levels of abstraction: 1) the analysis Phase; 2) the design Phase. Certain modules are responsible for each of them. The software structure and the functions of the modules are considered in detail by the authors. In conclusion, the authors present the syntax of the language used to describe the transformation rules of different patterns. This is similar to languages such as SQL, OCL, LINQ. To demonstrate the obtained results, the authors describe the test information system containing information about the patients of a hospital. The use chart (Use Case) shows the different categories of users and the types of the applied security

patterns. Then the structure of the template and the class diagram of the subject area after the application of this decision are illustrated in the form of a UML Class Diagram. This approach is applied to all selected templates, and the complexity of manual and automated applications is evaluated.

In [6], the model-oriented approach to the security applied in the information system of electronic voting is presented. The necessary security requirements, illustrated as the Use Diagrams of UML, were represented as functional requirements at the requirement formalization stage. After this, the authors describe the step-by-step algorithm for identifying and implementing the security requirements and then describe each key element in detail. The paper presents the application architecture and the main computing nodes (computers) which play a certain role. This allows the authors to determine possible vulnerability and attacks against which the system should be projected. The authors also present an approach to the security model implementation in the information system of electronic voting. The model is illustrated by the Sequence Diagram of language UML.

### III. THE MODEL OF ACCESS CONTROL

Currently, the classical model access control based on roles (Role-based access control, RBAC) has been widely used. Appeared in operating systems, it has the form presented in Fig. 1.

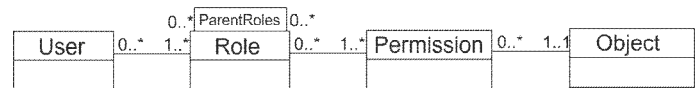


Fig. 1. Classical Role-based Access Control, (RBAC) model

This model is popular due to its plain architecture whose functions are as follows. The security system (model) creates multiple roles represented by the Role class. Each role is assigned certain access permissions represented by the Permission class. Permissions are assigned to different objects in the system, which is represented by the class Object. The user described by the User class is attached to at least one role. Moreover, these roles can be inherited, and this can simplify the process of assigning permissions to objects. This scheme is optimal for delineation of rights for objects of one type, for instance, for managing the permissions of access to file system objects (files, directories) in an operating system.

Software applications written in object-oriented programming languages require another security means because it are several types of objects that can be attributed by rights. For the optimal systems design the following optimality criteria (OC) for features are selected:

- 1. access rights for classes (OC1);
- 2. access rights for class properties (OC2);
- 3. access rights for objects (instances of classes) (OC3).



Fig. 2 shows the structure of an optimal model of access rights management for object-oriented applications.

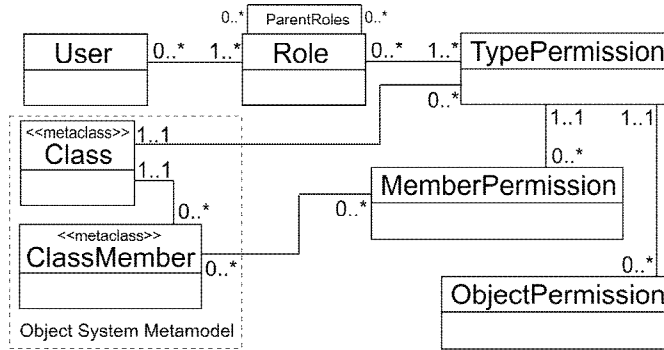


Fig. 2. Classical Role-based Access Control, (RBAC) model

We will examine this figure in more detail. To describe the objects which can be assigned the access rights, an advanced meta-model of the object system is used. In our case, it is enough to have information about the class and attributes (properties) of classes. To match the selected OC1, the class TypePermission which allows differentiating the access rights for the classes is designed. To differentiate the rights according to the properties of classes (see OC2), the class MemberPermission is introduced. The Class ObjectPermission is used to set permissions on the class copies corresponding to the OC3 requirement.

After clearing the structure and concept implementation, we begin to study of the final system. Figure 3 shows the implemented-by-authors model of access control for object-oriented applications in the form of class diagrams.

We will consider Figure 3 in more detail. All base classes implementing the key functionality of the security system have names ending by the suffix Base. So the SecuritySystemRoleBase and SecuritySystemUserBase classes form the root class for representing the roles of security and the system user, respectively. The TypePermissionMatrixItem class is used to specify the data type (class name) which needs the access rights. The following permission types are used for the classes:

1. AllowCreate allows the user to create objects (class instance);
2. AllowDelete allows the user to delete objects (class instance);
3. AllowNavigate allows the user to display a menu item to view the class instance;
4. AllowRead allows the user to view objects of the class;
5. AllowWrite allows the user to replace some objects of the class by other.

The class SecuritySystemMemberPermissionsObject allows one to describe the rights to some individual properties and to implement a complex security policy in which the user is prohibited from reading certain attributes of the class.

The class SecuritySystemObjectPermissionsObject is used to distinguish the rights between individual objects of the class

which satisfy some predicate. This condition holds in the property Criteria.

The UML diagram shows the relationship between associations which allows one to understand the relationship between classes. In the end, it should be noted that the developed security system allows an unlimited description of the types of access rights in an object-oriented system, which corresponds to the previously identified optimality criteria.

#### IV. EXAMPLES OF USING THE MODEL OF ACCESS CONTROL

To implement the above-described model of access control, it is very important to have the meta-information of the object system. The model is physically stored in a relational database according to the principles described in [7]. When designing a meta-model, the key challenge was to develop a hierarchy of meta-classes which allows one to save information about literal types and different classes of domain entities [8-10]. The design of the developed meta-model allows one to realize the subject-oriented approach to designing database applications for different fields [11-13]. In [14-16], the use of the metamodel in the design of information systems is described.

Then paper [17] describes the previously-used security model for access rights applied to an information system used to carry out scientific conferences. The model was repeatedly employed to manage the conference "Object system" (objectsystems.ru). Attention was paid to the security issues at the design stage. For this, the following roles were allocated to the users in the system:

1. **The organizer of the conference.** He is the main person and the user of the system. His responsibilities include the following tasks:
  6. to register the publications;
  7. to appoint the reviewer;
  8. to verify the corrections made by the authors according to the reviewer comments;
  9. to check the payments;
  10. to prepare the journal;
  11. to send the proceeding books and certificates to the authors of the papers.
2. **The author** writes an paper and sends it to the conference. The author's responsibility is also to revise the paper according to the reviewer's comments about the paper and, if necessary, to pay the registration fee.

3. **The reviewer** checks the author's paper and evaluates its quality. The review includes: to write a review indicating the observations and recommendations for its improvement; to formulate the review result (to accept the paper for publication or to reject it or to send it back for revision). During the preparation of the conference proceedings, the reviewers award nominations to the best papers submitted to the conference. However, in the general case, there are several reviewers.

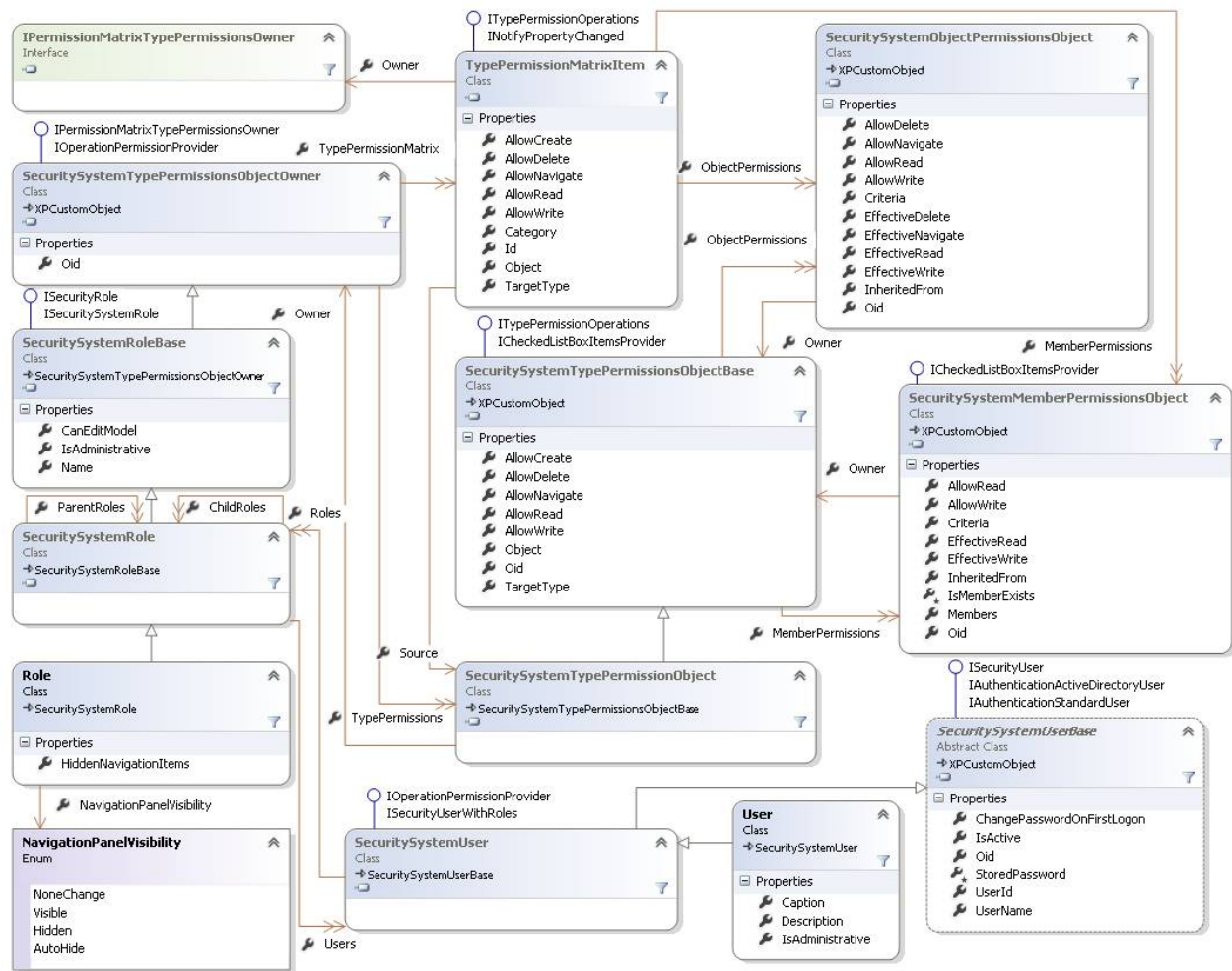


Fig. 3. UML class diagram of the implemented model of differentiation of access rights

On the basis of this information, classes and types of access are detected for different roles. Next, instances of classes presented in Figure 3 are created.

The paper [18] describes an information system of a beauty salon. Studying the business logic in this field shows that the system must implement a variety of different financial calculations determining the costs and profitability of the salon. This information can be presented only to the owner of the salon. The following roles are emphasized:

1. **Master.** Main task of the master is to provide services to clients. Therefore, each master can only view (read) the main system directories such as: Operating Schedule, Record/Visit, Schedule of visits, Customer, Leave/Sick leave/Compensatory leave/Absence, Service, Commodity, Certificate, Price, Interest, Master, master Category, room Category, Remnants of goods, Work schedule, Working hours;

2. **Salon administrator.** The main task of a manger is to monitor the activities of the salon. Namely, an administrator registers clients and monitors progress of master work. In the system, an administrator has right to add/edit/delete data from the directories: Visiting Schedule, Customer Master, work Schedule, Record/Attendance, Vacation/Sick

leave/Compensatory leave/Absence, Service, Commodity, Certificate, Discount, goods Receipt, Inventory, Price, Stock, Percent, client Category, master Category, service Category, Document, Movement of goods, remaining Stock, Sales, Salon, Working hours, Working time;

3. **Owner of the Salon** has all the same rights as the Administrator of the salon. In addition, he has right to view information from processed forms such as: Wages, Profit, and Profitability. The salon owner can also introduce new users in the system and add them only to the existing roles.

The papers [19-21] describe the information system architecture of fast food restaurants. The key feature of application of this class is that they are used in the places of public service with a large number of clients. In such software products, the critical maintenance time is very important, and so the graphical interface of the user must be ergonomic. The monoblocks with touch screens are often used as the hardware platform in such systems. Therefore, in such applications, attention is paid to the graphical interface of the user and to the principles of security settings. In this case, the following roles are selected:

- **Waiter.** The waiter's main task is to create purchase orders, to add the goods purchased by clients to the orders, and to arrange the payment;
- **Cashier.** A cashier cannot create new orders but can remove erroneous orders, view all orders issued in the current and previous shifts, and also issue the payment orders;
- **Manager.** His main task is to form consolidated reports on the work of a shift and to add new waiters and cashiers to the system;
- **Merchandiser.** The main task of the merchandiser is to introduce information about new food into the system.

When designing each of the above-described applications, the role of system administrator, who sets permissions for the existing roles and creates new roles, was also assigned. In fact, this role corresponds to the system administrator of a domain of the Windows operating system.

## V. INFORMATION SECURITY IN OA-SYSTEMS

The OA-approach to organization of the data structure and the computational process is currently being developed. The approach implements the object-oriented (OO) programming principle with a few other features [22,23,24]. The OA-approach requires new methods for the information security organization.

Unlike the OO paradigm, in OA, there is no distinction between the concepts of class and object. Instead of the class, a semantic network template, which is copied to generate a new semantic network, is used [25]. Also, there is no such a concept as the field of an object: a data and a program are represented as an information capsule (IC). Therefore, in the OA-system, the data security is focused on an information capsule (IC), and the OA-graph is protected through it. Let us explain it. The functional unit (FU) processes an OA-graph. Let us call it a processing FU. The processing FU usually takes reference to one of the IC (starting IC) of the OA-graph and produces a traversal from the IC. The traversal is performed as follows. A FU looks for the information pair (IP) in the IC with a specific attribute and goes by the link contained in its load to another IC of the OA-graph. Thus, the OA-graph security is provided through the security of the starting IC. Any other IC may be secured in the OA-graph similarly to the protection of the object field in the OO paradigm.

For the implementation of information security, a specialized FU, called the "Guard", is required. The functions of the FU are the control of the user accounts and roles (if the RBAC approach is used) and the creation and control of the access control list (ACL) for IC contained in the OA-graph. The Guard integrated to the processing FU controls the access permissions to a IC. The control is ensured as follows: operating FU before the analysis, the IC passes a reference to the access controller that checks the access permission to IC. If the access is denied, then the Guard blocks the FU performing the OA-graph traversal.

The access permissions information is stored in the ACL (fig. 4). The ACL can be attributed to the IC of the OA-graph by adding IP, called the security IP, with the attribute "ACL", the load of the IP contains a pointer to the ACL (one ACL can be assigned to one or several IC.). To prevent unauthorized access to the ACLs, the manipulation protection of security IP is included in the algorithm for controlling the processing FU: prohibition to remove the secure IP (the IP can only be removed during the removal of the IC, where the IP is located), prohibition to use the reference of the secure IP load, etc. The ACL is processed (creation, destruction and modification) by the Guard.

The proposed mechanism well emulates the protection class in the OO paradigm. If the secure IP is contained in the OA-graph, then when copying the OA-graph, the secure IP with the load containing the reference to the access rights matrix is copied too.

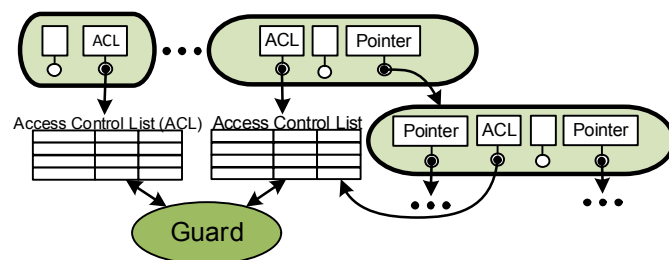


Fig. 4. The mechanism of data security in OA-computing system

The proposed methodology provides maximum flexibility of the security mechanism of the OA-graph and corresponds to all three criteria (OC1, OC2, OC3) applicable to the security of OO systems, i.e., protection of OA-graph (similar to object), a separate IC (similar to object fields), and OA-graphs copied from the OA-graph template (similar to the class protection). Moreover, all criteria are satisfied with a single protection mechanism.

## VI. CONCLUSIONS AND FURTHER RESEARCH

The above description shows that the established model of differentiation of access rights can successfully be used in applications in various domains, i.e. it is universal. Several applications where the security comes first are currently designed and implemented. This allows testing the proposed model completely and modifying it in accordance with the discovered drawbacks.

The model was developed in the OA-approach. The model is quite simple and satisfies all criteria for the security in the OO approach.

## References

- [1] Nagaratnam N., Nadalin A., Hondo M., McIntosh M., Austel P. Business-driven application security: from modeling to managing secure applications // IBM Systems Journal, Volume 44 Issue 4, 2005, 847-867 pp.
- [2] Xiao L., Peet A., Lewis P., Dashmapatra S., Saez C., Croitoru M., Vicente J., Gonzalez-Velez H., Lluich i Ariet M. An Adaptive Security

- Model for Multi-agent Systems and Application to a Clinical Trials Environment // 31st Annual International Computer Software and Applications Conference, COMPSAC 2007, 24-27 July 2007, Beijing, China, 2007, 261-268 pp.
- [3] Fengyu Zhao, Xin Peng, Wenyun Zhao. Multi-Tier Security Feature Modeling for Service-Oriented Application Integration // Eighth IEEE/ACIS International Conference on Computer and Information Science, ICIS 2009, 1-3 June 2009, Shanghai, China, 2009, 1178-1183 pp.
  - [4] Saleem M.Q., Jaafar J., Hassan M.F. Model Driven Security Framework for Definition of Security Requirements for SOA Based Applications // 2010 International Conference on Computer Applications and Industrial Electronics (ICCAIE), 5-8 Dec. 2010, Kuala Lumpur, 2010, 266-270 pp.
  - [5] Shiroma Y., Washizaki H., Fukazawa Y., Kubo A., Yoshioka N. Model-Driven Security Patterns Application Based on Dependences among Patterns // ARES '10 International Conference on Availability, Reliability, and Security, 15-18 Feb. 2010, Krakow, Poland, 2010, 555-559 pp.
  - [6] Salini P., Kanmani S. Application of Model Oriented Security Requirements Engineering Framework for Secure E-Voting // 2012 CSI Sixth International Conference on Software Engineering (CONSEG), 5-7 Sept. 2012, Indore, 2012, 1-6 pp.
  - [7] Oleynik P.P. Predstavlenie metamodeli ob'ektnoy sistemy v relyatsionnoy baze dannykh. Izvestiya vysshikh uchebnykh zavedeniy. Severo-Kavkazskiy region. Spetsvypusk «Matematicheskoe modelirovanie i komp'yuternye tekhnologii», 2005. - S. 3-8.
  - [8] Oleynik P.P. Organizatsiya ierarkhii atomarnykh literal'nykh tipov v ob'ektnoy sisteme, postroennoy na osnove RSUBD. Programmirovaniye, 2009, № 4. - S. 73-80
  - [9] Oleynik P.P. Implementation of the Hierarchy of Atomic Literal Types in an Object System Based of RDBMS // Programming and Computer Software, 2009, Vol. 35, No.4, pp. 235-240.
  - [10] Oleynik P.P. Class Hierarchy of Object System Metamodel. Object Systems – 2012: Proceedings of the Sixth International Theoretical and Practical Conference. Rostov-on-Don, Russia, 10-12 May, 2012. Edited by Pavel P. Oleynik. 37-40 pp. (In Russian), [http://objectsystems.ru/files/2012/Object\\_Systems\\_2012\\_Proceedings.pdf](http://objectsystems.ru/files/2012/Object_Systems_2012_Proceedings.pdf)
  - [11] Oleynik P.P. Domain-driven design of the database structure in terms of object system metamodel. Object Systems – 2014: Proceedings of the Eighth International Theoretical and Practical Conference (Rostov-on-Don, 10-12 May, 2014) / Edited by Pavel P. Oleynik. – Russia, Rostov-on-Don: SI (b) SRSPU (NPI), 2014. - pp. 41-46. (In Russian), [http://objectsystems.ru/files/2014/Object\\_Systems\\_2014\\_Proceedings.pdf](http://objectsystems.ru/files/2014/Object_Systems_2014_Proceedings.pdf)
  - [12] Oleynik P.P. Using metamodel of object system for domain-driven design the database structure // Proceedings of 12th IEEE East-West Design & Test Symposium (EWDTS'2014), Kiev, Ukraine, September 26 – 29, 2014, DOI: 10.1109/EWDTS.2014.7027052
  - [13] Oleynik P.P. Unified Metamodel of Object System. Object Systems – 2015: Proceedings of X International Theoretical and Practical Conference (Rostov-on-Don, 10-12 May, 2015) / Edited by Pavel P. Oleynik. – Russia, Rostov-on-Don: SI (b) SRSPU (NPI), 2015., [http://objectsystems.ru/files/2015/Object\\_Systems\\_2015\\_Proceedings.pdf](http://objectsystems.ru/files/2015/Object_Systems_2015_Proceedings.pdf)
  - [14] Oleynik P.P. The Elements of Development Environment for Information Systems Based on Metamodel of Object System. Business Informatics. 2013. №4(26). – pp. 69-76. (In Russian), [http://bijournal.hse.ru/data/2014/01/16/1326593606/1BI%204\(26\)%202013.pdf](http://bijournal.hse.ru/data/2014/01/16/1326593606/1BI%204(26)%202013.pdf)
  - [15] Oleynik P.P., Kurakov Yu.I. The Concept Creation Service Corporate Information Systems of Economic Industrial Energy Cluster. Applied Informatics. 2014. №6. 5-23 pp. (In Russian)
  - [16] Kurakov Y. I., Oleynik P. P. Implementation method a unified information system of economic production and energy cluster in coal industry // Mining information-analytical Bulletin (scientific and technical journal). 6 2015, pp. 260-273.
  - [17] Borodina N.E., Oleynik P.P., Galiaskarov E.G. Reengineering of Object Model by the Example of Information System for Cataloging Scientific Articles for International Conferences. Object Systems – 2014 (Winter session): Proceedings of IX International Theoretical and Practical Conference (Rostov-on-Don, 10-12 December, 2014) / Edited by Pavel P. Oleynik. – Russia, Rostov-on-Don: SI (b) SRSPU (NPI), 2014, 17-23 pp. (In Russian), [http://objectsystems.ru/files/2014WS/Object\\_Systems\\_2014\\_Winter\\_session\\_Proceedings.pdf](http://objectsystems.ru/files/2014WS/Object_Systems_2014_Winter_session_Proceedings.pdf)
  - [18] Kozlova K.O., Borodina N.E., Galiaskarov E.G., Oleynik P.P. Domain-Driven Design of Information System of a Beauty Salon in Terms of Unified Metamodel of Object System. Object Systems – 2015: Proceedings of X International Theoretical and Practical Conference (Rostov-on-Don, 10-12 May, 2015) / Edited by Pavel P. Oleynik. – Russia, Rostov-on-Don: SI (b) SRSPU (NPI), 2015. (In Russian), [http://objectsystems.ru/files/2015/Object\\_Systems\\_2015\\_Proceedings.pdf](http://objectsystems.ru/files/2015/Object_Systems_2015_Proceedings.pdf)
  - [19] Oleynik P.P, Yuzefova S.Yu., Nikolenko O.I. Experience in Designing an Information System for Fast Food Restaurants. Object Systems – 2014 (Winter session): Proceedings of IX International Theoretical and Practical Conference (Rostov-on-Don, 10-12 December, 2014) / Edited by Pavel P. Oleynik. – Russia, Rostov-on-Don: SI (b) SRSPU (NPI), 2014. – pp. 12-16. (In Russian), [http://objectsystems.ru/files/2014WS/Object\\_Systems\\_2014\\_Winter\\_session\\_Proceedings.pdf](http://objectsystems.ru/files/2014WS/Object_Systems_2014_Winter_session_Proceedings.pdf)
  - [20] Nikolenko O.I., Oleynik P.P, Yuzefova S.Yu. Prototyping and Implementation of Graphical Order Form for the Information System of Fast Food Restaurants. Object Systems – 2015: Proceedings of X International Theoretical and Practical Conference (Rostov-on-Don, 10-12 May, 2015) / Edited by Pavel P. Oleynik. – Russia, Rostov-on-Don: SI (b) SRSPU (NPI), 2015. (In Russian), [http://objectsystems.ru/files/2015/Object\\_Systems\\_2015\\_Proceedings.pdf](http://objectsystems.ru/files/2015/Object_Systems_2015_Proceedings.pdf)
  - [21] Pavel P. Oleynik, Olga I. Nikolenko, Svetlana Yu. Yuzefova. Information System for Fast Food Restaurants. Engineering and Technology. Vol. 2, No. 4, 2015, pp. 186-191., <http://article.aascit.org/file/pdf/9020895.pdf>
  - [22] P. B. Panfilov, S. M. Salibekyan Dataflow Computing and its Impact on Automation Applications. Procedia Engineering. Volume 69 (2014), Pages 1286-1295. URL: <http://www.sciencedirect.com/science/article/pii/S1877705814003671>
  - [23] Pavel P. Oleynik, Sergey M. Salibekyan. The Approaches to Implementation of Patterns of Static Object Models for Database Applications: Existing Solutions and Unified Testing Model. International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 24 (2015) pp 45513-45516.
  - [24] Salibekyan S.M., Panfilov P. B Object-Attribute Architecture is a New Approach to Object Systems Developing // Information technologies 2, 2012, pp 8-14
  - [25] Salibekyan S. M., Belousov, A. Yu., Graph Database Implemented by Object-Attribute Approach // Object systems – 2014 (winter session): materials of IX International scientific-practical conference (Rostov-on-Don, 10-12 may 2014) / ed. by P. P. Oleynik. - Rostov-on-don: SHI (f) SRSTU (NPI) to them. M. I. Platov, 2014. S. 70-76 URL: [http://objectsystems.ru/files/2014WS/Object\\_Systems\\_2014\\_Winter\\_session\\_Proceedings.pdf](http://objectsystems.ru/files/2014WS/Object_Systems_2014_Winter_session_Proceedings.pdf)

# Dynamic Key and Signature Generation According to the Starting Time

Andrey Kiryantsev

Volga Region State University of Telecommunications and  
Informatics Moskovskoe sh. 77, Samara, Russia  
Email: reyzor2142@gmail.com

Irina Stefanova

Volga Region State University of Telecommunications and  
Informatics Moskovskoe sh. 77, Samara, Russia  
Email: aistvt@mail.ru

**Annotation** – the article describes the algorithm of data encryption and digital signature algorithm. The keys are dynamically generated according to the starting time.

**Keywords** – cryptography, encryption, decryption, digital protection, digital signature, symmetric and asymmetric cryptosystems.

## I. INTRODUCTION

The necessity to secure information brings us to the basic concepts of cryptography: digital protection, digital signature and encryption. As you know, cryptography is engaged in search for solutions to such important security issues as confidentiality, identity verification, integrity and control of participants in the interaction.

Encryption is the process of converting data into a form which is not possible to read. It uses the encryption – decryption keys. The encryption process of the original message helps to ensure privacy by keeping information secret from someone it is not addressed to. A cryptographic system is formed by a set of conversion algorithms and keys used by these algorithms for encryption, key management system, as well as the original and the encrypted texts. Cryptosystems ensure the secrecy of transmitted messages as well as their authenticity and user's identity verification. The article offers new ideas for dynamic generation of keys and signatures depending on the starting time of the interaction between two subscribers.

## II. APPROACHES TO THE CONSTRUCTION OF CRYPTOSYSTEMS

There are two methods of cryptographic information processing with the keys – symmetric and asymmetric [1]. A symmetric (private) method implies that a sender and a receiver use the same key which they agree before the interaction for both encryption and decryption. If the key has not been compromised, then decryption database automatically verifies the sender, since it is only the sender who has the key which he/she can use to encrypt information, and it is only the recipient who has the key to decrypt the information.

The symmetric encryption algorithms use keys that are not very long and can quickly encrypt large amounts of data. Symmetric encryption systems have a common drawback – that is the complexity of the keys distribution. When an external party intercepts the key, the system of cryptographic protection will be compromised. When it is necessary to replace a key, it should be sent confidentially

to the participants of the encryption. Obviously, this method is not suitable when one needs to establish a secure connection with a large number of Internet subscribers. The main problems of this method are generation and secure transmitting of keys to the participants of the interaction. The question is what way it is better to establish a secure communication channel between the participants of interaction while sending keys through insecure communication channels. The lack of a secure key exchange method limits the expansion of symmetric methods of encryption in the Internet.

This problem is resolved in an asymmetric (public) encryption method. In an asymmetric system the document is encrypted with one key and decrypted with another one. Each participant of the information transfer generates two random numbers (private and public keys). The public key is transmitted through public communication channels to another participant of the encryption, but the private key is kept in secret. The sender encrypts the message with the public key of the recipient, and it is only the private key owner who may decrypt the message. This method is suitable for a wide usage. If each Internet user is assigned to his/her own pair of keys and the public keys are published as the numbers in the phone book, almost all users can exchange encrypted messages with each other.

All asymmetric cryptosystems are the object of direct attacks through the direct key enumeration, and, therefore, they must use much longer keys than those used in symmetric cryptosystems to provide an equivalent level of protection. This immediately affects the calculation resources required for encryption.

There is the necessity to verify that there is no distortion into the information in an e-document. Digital signature is used for this sake. Digital signature in a cryptosystem protects a document from changes or substitution and, thereby, guarantees its validity. It is a line, where the attributes of the document (for example, checksum of a file, etc.) and its contents are encoded, so that any change in the file even with the unchanged signature may be detected. When a document is protected by a digital signature, it verifies the document itself along with the private key of the sender, and the recipient's public key. The owner of a private key is the only one who can sign the document correctly. To verify the digital signature of the document, the recipient uses the sender's public key. No other key pair is suitable for verification. Thus, unlike an ordinary signature, digital signature depends on the document and the



sender's public key. Therefore, it is several times safer than an ordinary signature and a seal.

Despite the fact that digital signature certifies the authenticity of the document, it does not protect it from unauthorized reading. Both symmetric and asymmetric encryption systems have their advantages and disadvantages. The shortcomings of symmetric encryption are in the complexity of replacing a compromised key, and the disadvantages of asymmetric encryption are in a relatively low speed of work.

These problems are addressed to the encryption systems that use the combined algorithm, which enables high-speed encryption and sending of the encryption keys through the public channels. In order to avoid low-speed of asymmetric encryption algorithms, a temporary symmetric key is generated for each message. The message is encrypted with a temporary symmetric session key. Then this session key is encrypted with a public asymmetric key of a recipient and an asymmetric encryption algorithm. Due to the fact that a session key is much shorter than a message itself, the time of encryption will be relatively short. After that this encrypted session key is transferred to the recipient along with the encrypted message. The recipient uses the same asymmetric encryption algorithm and his/her private key to decrypt the session key and the received session key is used to decrypt the message.

The mentioned above makes it obvious that integrated encryption algorithms currently have a promising line of development in modern cryptosystems.

### III. ALGORITHM DESCRIPTION

It is time to consider the operation principle of the suggested method to data encryption with a session symmetric key, generated at the moment of interaction between two subscribers. A session key is encrypted with the exposed asymmetric key of the recipient and Diffie-Hellman's algorithm [2]. The algorithm allows two sides to get common private key through the channel that is unprotected from discreet listening, but it is protected from the channel substitution. The received key can be used for message exchange through symmetric encryption.

Diffie-Hellman's algorithm uses one-sided function  $F(X)$  with two attributes:

- there is a polynomial algorithm of values  $F(X)$ ,
- there is not a polynomial algorithm of inverted function  $F(X)$ .

To put simply, this function doesn't include decryption of the encrypted text.

Figure 1 presents encryption's block diagram according to the Diffie-Hellman's algorithm.

The function with a secret is the function  $Fk$ ; it depends on  $k$  and has the following properties: there is a polynomial algorithm of calculation  $Fk(X)$  value for any  $k$  and  $X$ , and there is not a polynomial algorithm of the inverted  $Fk$  for unknown  $k$ ; but there is a polynomial algorithm of inverted  $Fk$  for the known  $k$  parameter.

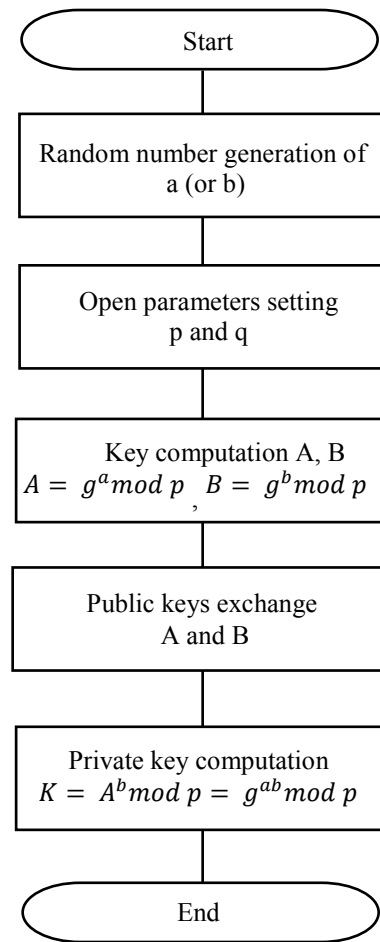


Fig. 1. Block diagram of Diffie-Hellman algorithm

The algorithm operation is presented in the following example. Andrew defines variables  $g$  and  $p$  which are large numbers. And he also conceives his private number  $a$  and calculates the value  $A$  using the formula

$$A = g^a \mod p \quad (1)$$

Then he transmits it to Natasha along with the conceived values of  $g$  and  $p$ . Natasha conceives her private number  $b$ . Through the same formula as Andrew does, she calculates her public number

$$B = g^b \mod p \quad (2)$$

and sends to Andrew. It is possible that the malicious user can get both values, but he will not modify them as he is unable to interfere in broadcasting process.

At the second stage Natasha calculates the value of  $K$  having number  $B$  and the received number  $A$ :

$$K = A^b \mod p = g^{ab} \mod p, \quad (3)$$

This is the key for encryption. Then, Andrew calculates his key using number  $B$  received from Natasha and his calculated number  $A$

$$K = B^a \mod p = g^{ab} \mod p. \quad (4)$$

You can see in examples (3) and (4) that Andrew gets the same number  $k$ , as Natasha. As a result, there is a root key that will be used in generating temporary key and message's signature in the future.

If the root key is used as a private key, a malefactor will be forced to meet with a practically undecidable (for a reasonable period of time) problem of calculating the number  $g^{ab} \bmod p$  having numbers  $A = g^a \bmod p$  and  $B = g^b \bmod p$ , intercepted in the public channel if  $p$ ,  $a$  and  $b$  are large enough numbers.

Now it is time to explain the process of temporary key generation. It follows the same HMAC (hash-based message authentication code) algorithm [3] and its standard RFC2104. According to them, information integrity is verified with private key. This standard allows to ensure that transmittable or stored at unreliable environment data were not change by unknown persons.

The HMAC algorithm contains the standard, describing the process of data exchange, the process of data integrity verification with the help of private key and hash-function. Depending on the hash-function we can distinguish HMAC-MD5, HMDC-SH1 etc.

In the article the hash-function is generated from the root key by the suggested algorithm, for example: md5 (rootKey + Time). Function md5 is a modification of hash-function MD5. While generating hash-function the time, particularly its second value, will be rounded. As it is known, time is presented in the format HH:MM:SS and rounding happens in the last format's unit. If there are more than 30 sec. in the value of starting time SS, then they are rounded upward, if there are less than 30 sec., then they are rounded downward. The message will be encrypted exactly with this key, and also through this algorithm one can generate digital signature of a message to verify the message. As a result we get a resistant system of dynamic keys for messages encryption and signature, where participants do not need to exchange some data for generation and root key generally.

A generalized algorithm of messages encryption in cryptosystem with the key and signature generation is presented on Figure 2.

#### IV. WORKING PROTOTYPE

Web technologies and JavaScript language were chosen for prototype realization. Due to it, the program will become a cross-platform and can be loaded everywhere, when there is a support of JavaScript specification (EcmaScript 5) and HTML 4 support. The JavaScript language was not chosen randomly, as at this moment it is the only "native" language for browser and it is supported by all browsers on default.

Below we can consider fragments of prototype code as an example.

Mass Math.random is used for generation of a large number  $p$  with Diffie-Hellman algorithm.

This approach is justified by the fact that the JavaScript language cannot work with large numbers (BigInt), as the algorithm requires it.

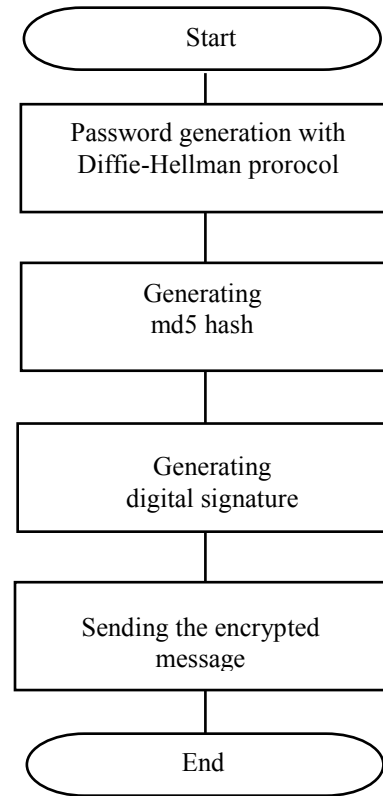


Fig 2. Block diagram of encryption by the key and signature generation algorithm

The code generation example of a large number in JavaScript language looks as this:

```

random(1000000000,9999999999) + " +
random(1000000000,9999999999) + " +
random(1000000000,9999999999) + " +
random(1000000000,9999999999) + " +
random(1000000000,9999999999) + " +
random(1000000000,9999999999) + " +
random(1000000000,9999999999) + " +
random(1000000000,9999999999);
  
```

Then the code of message's generation to JavaScript language looks as this:

```

$scope.getSign = function(){
    return md5($scope.msg + $scope.username +
    bigInt2str(a_sec, 10).toString() + datetime);
}
  
```

Function md5(arg) returns the hash line from argument arg. Function bigInt2str is a function that allows to work with large numbers in JavaScript. \$scope.username allows to insert a username. In this way we get a unique signature for each user. There is a screenshot of text values' substitution and the result of the performed program:



md5 ( текст d1r31q12 12:23 24.11.15 )

key                      time

text

5c733932c8910c2c6cb1432d1eb6f117

The time test of script was conducted through the prototype. In this test the following e-devices were used:

- 1) The computer – INTRL i5 (Windows 10/chrome)
- 2) The phone – Nexus 5 (android 6.0.1/ chrome)
- 3) The phone – Samsung galaxy ace (android 4.2.2/ browser).

In table 1 the results of the algorithm individual steps are provided. The steps are applied in different application. In Figure 3 there is a diagram that visualizes experiment results. From the table analysis it is obvious that the algorithm works very fast on the mobile phones

Hash-function algorithm MD5 is not selected occasionally, it is the fastest, the most common one. It has the simplest hashing algorithm that may be used for signature generation. Besides MD5 possesses a very interesting property. For instance, if at least one byte in a line is changed, the view of the resulting hash line will change dramatically.

Table 1. Time of algorithm application in different devices at different stages (msec).

	INTEL i5 (Windows/chrome)	Nexus 5 (android 6.0.1/chrome)	Samsung galaxy ace (android 4.2.2/ browser)
Diffie-Hellman generation	20,915	166,706	220,53
MD5 generation	0,81	3,315	6,21
Sign generation	0,27	0,48	0,72
Total time	22,883	170,89	229,416

The logic of the encryption algorithm can be considered in five steps. After the data are received there is the process of preparing the data flow to the calculations.

Step 1. First, the flow line requires alignment for hashing. At the end of the stream one on-bit and the necessary number of off-bits are registered. After the input data alignment, the length of the stream should be equal to  $512 \cdot N + 448$ .

Step 2. At the end of the message one should add 64-bit result for alignment. There are 4 low-order bits which are put first, and then high-order bits follow. If the stream length exceeds  $2^{64} - 1$ , only low-order bits are written down. After that, the stream length becomes 512-fold. The calculations are made with data flow presented as an array of 512-bit words.

Step 3. Then it is necessary to initialize four 32-bit variables (A, B, C, D) and to set their initial values with hex numbers: "low-order byte comes first". For example,

A = 01 23 45 67; // 67452301h

B = 89 AB CD EF; // EFCDA89h

C = FE DC BA 98; // 98BADCFEh

D = 76 54 32 10. // 10325476h

The results of intermediate calculations will be stored in these variables. Then it is time to initialize constants and functions required in further calculations

Four laps will require 4 functions with the logical operators XOR ( $\oplus$ ), AND ( $\wedge$ ), OR ( $\vee$ ), NOT ( $\neg$ ):

$$FunF(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z),$$

$$FunG(X, Y, Z) = (X \wedge Z) \vee (\neg Z \wedge Y),$$

$$FunH(X, Y, Z) = (X \oplus Y \oplus Z),$$

$$FunI(X, Y, Z) = Y \oplus (\neg Z \vee X).$$

The 64-element table of invariables is structured as follows:

$$T[n] = \text{int}(2^{32} \cdot |\sin(n)|)$$

Each 512-bit block of the flow passes through 4 stages of calculation, 16 laps each. For this the block is presented as an array X of sixteen 32-bit words. All the laps are of the same type, but they differ in the rotate shift by  $s$  bits of a 32-bit argument. The number  $s$  is defined for each lap.

Step 4. Steps in loop calculations. Put  $n$  element into the block from an array of five 12-bit blocks. The values A, B, C, D, remain after operations with the previous blocks (or their values in case the array goes first).

AA = A

BB = B

CC = C

DD = D

Sum the values with the result of the previous loop:

A = AA + A

B = BB + B

C = CC + C

D = DD + d

After the loop ends, check if there are any blocks for calculations left. If there are some, go to the next array element ( $n+1$ ) and the loop repeats.

Step 5. The result of the hash-function calculation is formed in ABCD buffer. If the result starts with the low-order byte A, one gets MD-5 hash.

Figure 4 presents a screenshot of md5 hash function working prototype in the CRYPT2CHAT app [4]. It resorts to a modified MD5 hash function.



Fig 3. Histogram of algorithm performance time by different e-devices

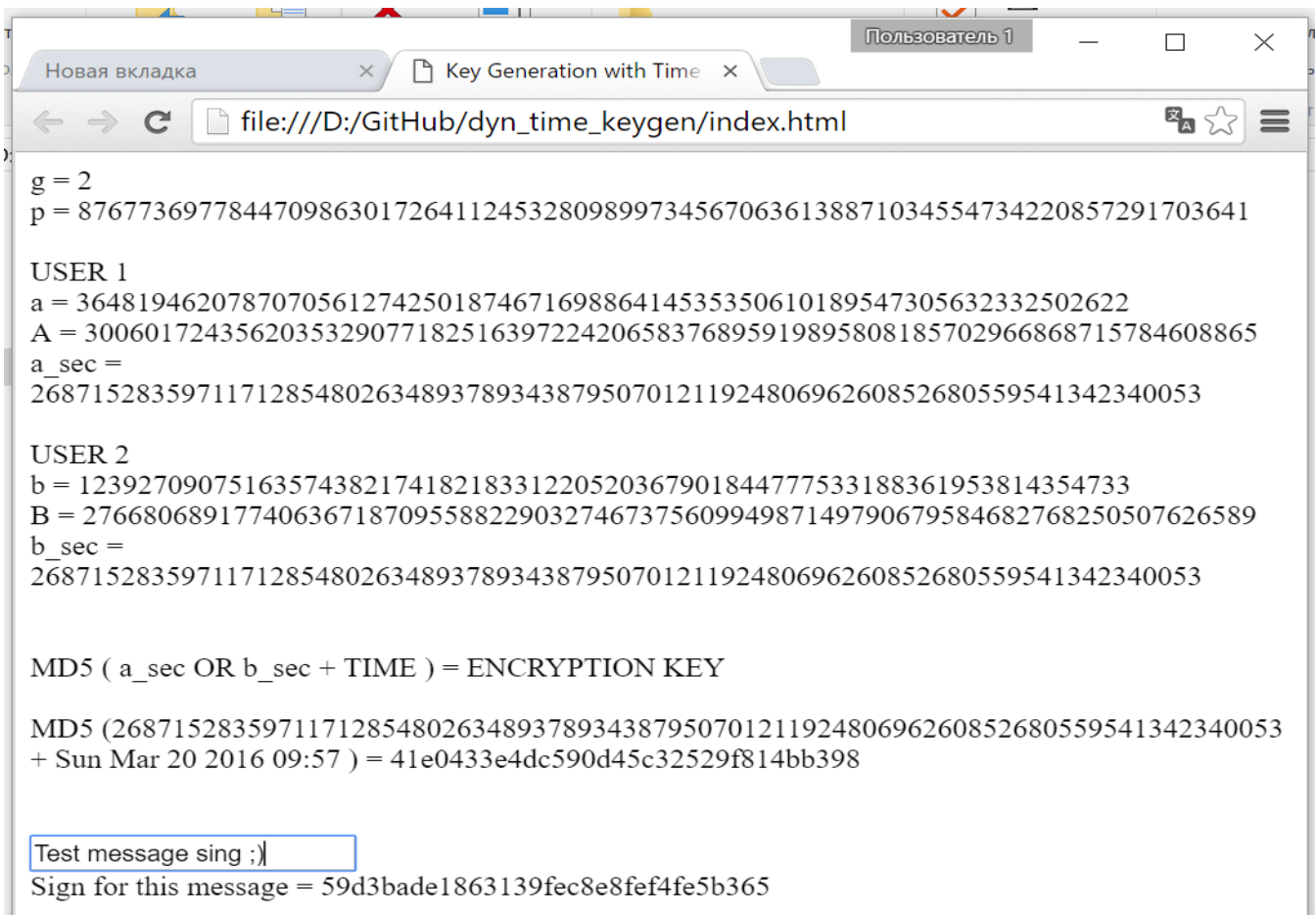


Fig 4. Prototype of Application work

## V. EVALUATION OF ALGORITHM EFFECTIVENESS

The cryptographic strength of the proposed algorithm for keys and signatures generation depends on the encryption method that combines the algorithms of symmetric and asymmetric encryption.

The cryptographic strength is a quantitative characteristic of encryption algorithms – intrusion into a particular algorithm requires a certain number of resources. This is the amount of information and time required to perform the attack, as well as the memory required to store information used in the attack.

An attacking encryption algorithm typically aims at solving the following tasks:

- to get public text version from the encrypted one,
- to calculate the encryption key.

The second task is usually more challenging than the first one. However, having the encryption key the cryptanalyst can later decrypt all the data encrypted with a key.

The algorithm is considered to be secure if a successful attack at it requires from an attacker unattainable calculating resources in practice, or open intercepted and encrypted messages, or if decryption is so time-consuming that currently protected information would lose its relevance. In most cases, the cryptographic strength cannot be mathematically proven, you can only prove the vulnerability of the algorithm or calculate the time required to find a key. For this sake one should take into consideration the difficulty of a given mathematical problem that serves as the basis for the encryption algorithm.

To estimate the time of password configuration to gain an unauthorized access to the channel of two subscribers, we have the equation [5]:

$$t = \frac{N^0 + N^1 + N^2 + N^L}{V} \quad (5)$$

It estimates time in the worst case. Here  $t$  is the time required for the guaranteed password configuration,  $V$  is the number of combinations per second in brute search,  $N$  is the number of characters in the configured password,  $L$  is the length of the password.

In case with md5 algorithm the number of characters is 36. This number includes 26 symbols-letters in the Latin alphabet (a...z) and 10 symbols of Arabic numerals (0..9). The number of symbols in the secure key for encryption or signing is 32. To calculate speed of the brute symbol search, we'll take an intel i7 and a video card Radeon HD5850 1024 MB. Their power equals to 65 000 passwords per second, calculated empirically.

As a result of substitution of values in (5) the estimated time will be:

$$t = \frac{36^0 + 36^1 + 36^2 + 36^{32}}{6500} = 9.745 \times 10^{44} \text{ c.}$$

Converting the seconds into a larger value, we get the result  $3.09 \times 10^{37}$  years.

Conclusion: this algorithm can be considered secure from attacks and encryption key calculation, as the time for the key search outweighs the actual time of work with data.

In sources [5, 6] an algorithm of dynamic key generation is offered. It is presented as a self-authenticated method with timestamp. In the patent the author employs asymmetric encryption-decryption algorithm. In contrast in this article the described algorithm is symmetric. This helps exclude sending and receiving any key, which increases security of data transmission. Moreover, Google team uses slightly similar algorithm of key generation. However, its development group employs another hash function that is not connected with encryption. Additionally, password configuration is a part of the algorithm that we provide.

## CONCLUSION

The algorithm for temporary keys and signatures generation can be used to teach students the basics of cryptography, and used in real projects. Coupled with a VPN or TOR networks it becomes more secure due to the new encryption level [7].

## REFERENCES

1. Mikhail Adamenko. The basics of classical cryptology. The secrets of ciphers and codes. Publishing DMK. p. 2014 - 256.
2. Diffie, W. and Hellman, M. E. *New directions in cryptography*, 1976.
3. Maurer U.M, Wolf S. The Diffie-Hellman Protocol. Retrieved. Designs, Codes and Cryptography, 2000. T. № 2-3. p.147-171.
4. The construction of the password generator. Retrieved from [www.scribub.com/limba/rusa/194620205.php](http://www.scribub.com/limba/rusa/194620205.php), 2013-08-02 (accessed February, 2016).
5. Self-authenticated method with timestamp. Patent US 20140325225 A1. Retrieved from <http://www.google.com/patents/US20140325225> (accessed Oct. 30, 2014).
6. SELF-AUTHENTICATED METHOD WITH TIMESTAMP - DIAGRAM, SCHEMATIC, AND IMAGE 06. Retrieved from [http://www.faqs.org/patents/imgfull/20140325225\\_06](http://www.faqs.org/patents/imgfull/20140325225_06) (accessed Oct. 30, 2014 Sheet 5 of 5).
7. Kiryantsev A. C., Stefanova I. A. Constructing Private Service with CRYPT2CHAT application // Proceedings of the Institute for System Programming of RAS, Volume 27. Issue 3. 2015. p. 279-290.

# Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS

Casper Thule  
Department of Engineering  
Aarhus University  
Aarhus, Denmark  
Email: casper.thule@eng.au.dk

Peter Gorm Larsen  
Department of Engineering  
Aarhus University  
Aarhus, Denmark  
Email: pgl@eng.au.dk

**Abstract**—The development of Cyber-Physical Systems often involves cyber elements controlling physical entities, and this interaction is challenging. It can be useful to create models of the constituent components and simulate these in what is called a co-simulation, as it can help to identify undesired behaviour. The Functional Mock-up Interface describes a standard for constituent components participating in such a co-simulation. This paper describes an exploration of whether different concurrency features (actors, parallel collections, and futures) increase the performance of an existing Co-Simulation Orchestration Engine performing co-simulations. The analysis showed that concurrency can be used to increase the performance in some cases, but in order to achieve optimal performance, it is necessary to combine different strategies.

## I. INTRODUCTION

Cyber-Physical Systems (CPSs) need to have close interaction between computer-based *cyber* parts controlling *physical* artefacts in a dependable way. In order to develop CPSs in a dependable manner it can be useful to create *models* of constituent components that jointly form the system. A constituent model is an abstract description of a constituent, where the irrelevant details are abstracted away. Constituent models can be described in very different forms depending upon their nature, but here we will restrict ourselves to Discrete Event (DE) and Continuous-Time (CT) models representing very different disciplines. Such constituent models can then be used in a collaborative simulation (a *co-simulation*), which is able to couple models created in different formalisms. Thereby it is possible to simulate the entire system by simulating the components and exchange data as the common simulated time is progressing.

Typically such co-simulations are organised with a master-slave architecture where a Master Algorithm (MA) is used to manage the simulation. Figure 1 shows an example of four slaves, their dependencies, and input/output ports. It is the responsibility of the MA and thereby the master to orchestrate the simulation. This means to allow the different slaves to progress for determined time steps and resolve the dependencies between steps. A co-simulation often consists of three phases: Initialisation, simulation, and tear down. In the initialisation phase the master gets the properties of the slaves, chooses an MA, initialises the slaves, and establishes the communication channels. Next, in the simulation phase

the master retrieves output values from the slaves, sets input values on the slaves, and invokes them to run a simulation step with a specific time step size. The slaves must respond with a status whether the step was accepted. In this phase, it can be necessary to perform a rollback<sup>1</sup> (if possible) for the relevant slave and run the simulation again with a different step size. Lastly, the outputs from the slaves are retrieved and the process repeats until a configured end time is reached. The final phase is tear down, where the slaves are shut down, memory is released, results are reported and so forth.

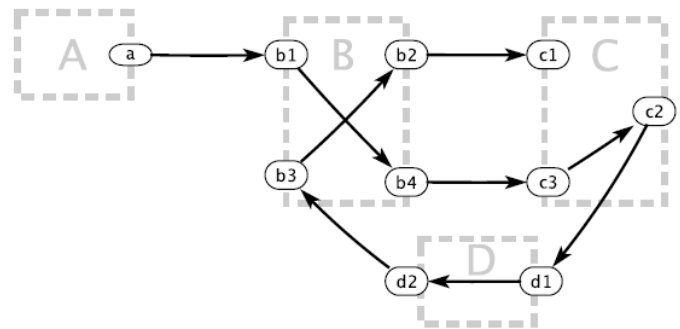


Fig. 1. Example of a simulated CPS with dependencies between slaves (the gray boxes) via their respective ports (the black ellipses) [1].

A challenge in using co-simulation as part of developing CPSs is that many complex multi-disciplinary systems cannot be modelled naturally in one simulation tool alone, but require several specialised simulation tools, that each do their part [2]. This makes it necessary to develop solutions tailored for a specific purpose instead of generalised solutions, which is expensive.

The Functional Mock-up Interface (FMI) was created to solve these challenges, as it is a tool-independent standard for co-simulation [3]. The standard provides and describes C interfaces, that can be partly or fully implemented by a component, which is then called a Functional Mock-up Unit (FMU). This makes it possible to create generalised solutions, as the components can contain their own solvers, and still

<sup>1</sup>A rollback can be necessary e.g. if a slave rejects a step size.

adhere to FMI. The INTO-CPS project<sup>2</sup> [4] makes use of FMI for a simulation kernel of a tool suite ranging from original requirements expressed in SysML over heterogeneous constituent models that can be co-simulated and gradually moved down to their corresponding realisations.

When developing CPSs using co-simulation, it is desirable to execute the simulations as fast as possible to enable the use of increasingly complex models and try a greater range of test scenarios. As many processors today have multiple cores [5] concurrency may increase the performance of an application, but it also introduces overhead. It is therefore of keen interest to determine, how concurrency can be used to potentially improve the performance. The performance in this context is considered to be how fast a co-simulation is performed, and is therefore measured in terms of time.

This paper describes how the usage of concurrency was implemented in an existing application called the Co-Simulation Orchestration Engine (COE), which orchestrates co-simulations using FMI. Different implementations were performed in Scala using three different concurrency features: Akka Actors [6], futures [7], and parallel collections [8]. These were chosen because they offer different capabilities that can be taken advantage of in the COE, and therefore the trade-off between features and performance is interesting. One of the most important capabilities is composability, because FMUs can have different step sizes and rollbacks can be necessary, which can lead to complicated scenarios. Following is a short description of the concurrency features:

**Parallel Collections:** The motivation behind adding parallel collections to Scala was to provide a familiar and simple high-level abstraction to parallel programming [8]. Parallel collections are conceptually simple to use, as a regular collection can be converted to a parallel collection by invoking the function “par”. Once it is a parallel collection, functions such as map and filter are executed concurrently. Parallel collections are considered less composable than the other implementations, as the results are gathered in a blocking fashion.

**Futures:** A future is a placeholder for a value, that is the result of some concurrent calculation, and it can be accessed synchronously or asynchronously. The term “future” was originally proposed by Baker and Hewitt [9] in the context of garbage collection of processes. As opposed to parallel collections, it is possible to chain futures, such that when a future has been computed, the computed value is passed to the chained future.

**Actors:** The Actor Model was introduced as an architecture to efficiently run programs with a high degree of parallelism without the need for semaphores [10]. An *actor* is an autonomous object that encapsulates data, methods, a thread, a mailbox, and an address [11]. Actor methods can return futures, and therefore offer the same composability as futures in this regard. Actors also provide additional composable features, such as hierarchical structures, remote capabilities,

message parsing, and so on.

The paper is structured as follows: Section II describes the initial implementation and the implementations using concurrency. Afterwards, Section III describes how the implementations were tested and presents the results. Then related work is treated in Section IV. Lastly, the work is summarised in Section V and the future work is outlined in Section VI.

## II. CO-SIMULATION ORCHESTRATION ENGINE IMPLEMENTATIONS

This section concerns the implementations of the COE application<sup>3</sup>. It focuses on the MA part of the implementations, as the initialisation and tear down phases are unaltered for the implementations described below.

The COE application runs as a web server using HTTP. The following HTTP requests are performed in the given order to run a simulation:

**Initialise:** A configuration file is sent to the web server. The configuration file contains the FMUs to be used in the simulation, the mapping between input and output values, and whether to use a fixed or variable step size.

**Simulate:** This request starts a simulation.

**Results:** This request returns the result and duration of a given simulation.

There are different implementations of the MA in the COE: A sequential implementation, and three implementations that execute concurrently, as described above. These different implementations were developed in order to test and compare the performance of the COE in a sequential/concurrent setting and determine whether using concurrency could improve the performance.

### A. Sequential Implementation

The sequential implementation of the MA consists of the following steps in the given order:

**Resolve inputs:** This step consists of mapping the outputs of the FMUs to the inputs of the other FMUs.

**Set inputs:** The input values determined in the previous step are passed to the FMU instances in this step.

**Serialize state:** In this step the states of the FMUs are serialized, so it is possible to perform a rollback in case of an error.

**Get step size:** If variable step size is supported by the FMUs, then the maximum step size is retrieved in this step. Otherwise a configured fixed step size is used.

**Do step:** The FMU instances are invoked to perform a step with the step size determined in the previous step. This function contains the most extensive calculations performed by the FMUs.

**Process result:** The return values from the previous invocations are analysed and in case of any errors a roll back is performed or the simulation is terminated.

**Get state:** The state in terms of output values is retrieved in this step, and thereby the next iteration can begin.

<sup>2</sup>Public deliverables and more information regarding the INTO-CPS project can be retrieved from <http://into-cps.au.dk>.

<sup>3</sup>See [12] for further details on the implementation.

In the sequential implementation a mapping operation is performed over the FMU instances in every step except the “Process result” step, where it depends on whether errors are encountered and possibly which errors. This sums to six, possibly seven, mapping operations over the FMU instances.

### B. Implementations with Concurrency

When implementing concurrency in the COE it is desirable that as much work as possible is performed in every concurrent invocation. To allow for a better usage of concurrency some functions should be grouped, such that a group of functions can be invoked concurrently. If concurrency was used in the sequential implementation to invoke the FMUs without refactoring the implementation, it would be necessary to invoke every step in different concurrent invocations. This would result in several thread initialisation and synchronizations per simulation step, where a synchronization is a waiting operation until all threads have finished computing. An example of this is shown in Figure 2. The figure shows a possible usage of concurrency based on the sequential implementation with four FMUs (black frame), where the functions “Set inputs”, “Serialize state”, “Do step”, and “Get state” are invoked in different concurrent invocations. The realised implementation (orange frame) invokes the functions using the same concurrent invocation for a given FMU. This will be described further below.

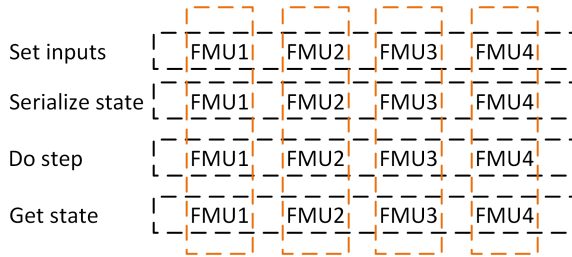


Fig. 2. The orange frames represents a possible usage of concurrency based on the sequential implementation. The black frames represents the usage of concurrency based on the implementations.

By refactoring and grouping these functions, it is possible to reduce the thread initialisations and synchronizations. This leads to more work performed by every spawned thread and fewer synchronizations, which minimizes the overhead of using concurrency. It is not possible to eliminate synchronization completely, because it is necessary to resolve the inputs for the FMUs before progressing, which requires retrieving the outputs from other FMUs, and therefore the simulation cannot continue until this has been performed. Besides minimizing the overhead of using concurrency, this grouping will also help to minimize the number of mapping operations performed in the steps in the sequential implementation, which is desirable to improve the performance.

The grouping and flow of a simulation step for the implementation using concurrency is shown in Figure 3. The grouping was implemented in a separate and encapsulated function that exhibits referential transparency to prevent the necessity

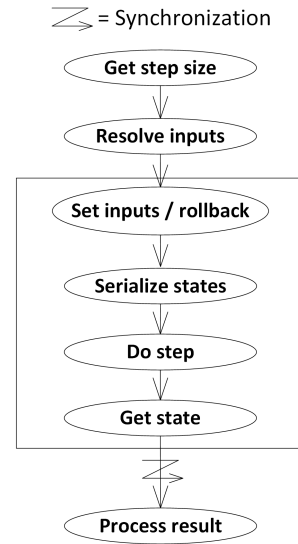


Fig. 3. Simulation step flow in the implementation using concurrency. The box represents the functions grouped together.

of locking mechanisms. This grouping will be referred to as the concurrent entity below.

By creating these concurrent entities, it was a conceptually simple task to take advantage of the concurrency features. Furthermore, it effectively reduced the mapping operations from six, possibly seven depending on the step “Process result”, to three. This implementation also makes it possible to include “assignment functions” such as “Set inputs” in the concurrent entity without lowering performance. Including “Set inputs” as its own concurrent invocation (as shown in the black frame in Figure 2) would lower the performance, because the overhead of using concurrency is too high compared to invoking the function sequentially. Using the grouping (the orange frame in Figure 2) it improves performance to include “Set inputs”, because it can be grouped with the other functions, e.g. “Do step”, without additional overhead. However, the grouping also came with a trade-off: In the sequential implementation, the state would not be retrieved, if one or more FMUs fail in the step called “Do step”, because it would be wasteful due to the error(s). But in the implementation using grouped functions, the state of the FMUs not failing in the step “Do step” would still be retrieved, because the entities responsible for the FMU simulation step are unaware of the state of other entities until the synchronization phase<sup>4</sup>. This can therefore lead to unnecessary retrieval of states.

In the sequential implementation, the flow is to calculate the parameters necessary for the next immediate function to be invoked on the FMUs, and then calculate the parameters again. In the implementations with concurrency this is changed to calculate the parameters necessary for an entire simulation step, and invoke the concurrent entity for each FMU concur-

<sup>4</sup>Several programming languages offer the possibility to abort threads in a case like this. However, that increases the complexity and is not considered applicable in general.

rently. This makes it possible to maximize the workload for each concurrent invocation.

### III. TESTING

This section presents the evaluation of the COE described in Section II. The purpose is to gain data that can be used to compare performance of the sequential implementation and the implementations using concurrency. Furthermore, as concurrency can lead to non-determinism, it is important to verify the simulation results, which are the output values of the FMUs at different points in time relative to the step sizes. For this purpose, the sequential implementation was considered an oracle, and therefore simulation results of the concurrent implementations were compared against simulation results from the sequential implementation. In the longer term the plan is to use a representation of the FMI semantics as the ultimate oracle [13]. Here semantics is provided using the Communicating Sequential Processes [14] and this has been used to model check FMI for deadlock and livelock properties using the FDR tool [15].

The following test principles were followed during testing:

**Test environment:** A test consisting of multiple simulations should be performed on the same hardware with approximately the same processes running during the test. The reason for stating “approximately the same processes” is, that the tests were run in a Windows environment, where it is not possible to completely control the running processes from the Operating System. All processes irrelevant to the execution of tests should be disabled during the tests.

**Test functions:** To limit inconsistencies in the processes running between simulations, each test should be implemented as a single test function. This means that a test performing simulations using the sequential implementation and the three concurrent implementations should be implemented in one test function to avoid undesirable interaction required to start other tests. To further ensure usable results the COE application should be restarted for every simulation.

**Correct simulation results:** The sequential implementation is considered to be an oracle and it is assumed that it calculates the “correct” simulation results. It should be verified that the concurrent implementations calculate the same simulation results as the sequential implementation.

**Automation:** The tests should be automated so they are easy to replicate and less prone to manual errors. This will also make them usable in the future development of the COE.

#### A. Test Setup

To enable automatic testing a framework was developed. This enabled testing of different concurrent implementations, evaluation of performance, and verification of consistency between the sequential simulation results and the concurrent simulation results. Implementation-wise this required support for launching the different implementations with different arguments, invoking the web servers using HTTP requests along with gathering, and verifying the consistency of results. To verify the consistency of results, the simulation results

TABLE I  
RESULTS FROM HVAC #1

Sequential	Future	Par	Actor
31256	29822	31980	30919

TABLE II  
RESULTS FROM SI #1

Wait	Sequential	Future	Par	Actor
0.0	195	330	656	374
0.5	4468	4635	5161	4715
1.0	8758	8938	9545	9032

of the implementations using concurrency are automatically compared to the simulation results of the sequential implementation, as this is considered an oracle.

Different FMUs were used in the tests to investigate the performance, including a configurable FMU, that was developed to control the level of computations, which will be described below. The tests and their corresponding FMUs are the following:

#### Heating, Ventilation, and Air Conditioning (HVAC) test:

This test uses FMUs that perform the most extensive computations available in the project. The simulation consists of five FMUs: one controller FMU and four Fan Coil Unit FMUs. A test, which will be referred to as HVAC #1, was set up with an end time of 1000 seconds and a step size of 0.1 seconds.

**Sine Integrate Wait tests:** These tests consist of three different FMUs, that perform limited computations, and therefore one has been modified. The FMUs are: a sine FMU generating a sine wave, an integrate FMU that integrates the sine values, and a modified integrate FMU. It is possible to configure the modified integrate FMU, such that it performs busy waiting in the “Do step” function for a given number of microseconds. It makes use of “QueryPerformanceCounter” recommended by Microsoft to use when high-resolution time stamps are required with microsecond precision [16]. The configuration of the busy wait does not have any impact on the performance of the FMU, because it happens in the initialisation phase, which is not part of the performance measurement. These FMUs were used to set up three tests, referred to as SI #1/2/3, where each simulation in the tests have an end time of 100 seconds and time step size of 0.1 seconds. The tests are the following:

SI #1 consists of one sine FMU, one modified integrate FMU, and three simulations: In the first simulation, the modified integrate FMU has a wait time of zero milliseconds, then 0.5 milliseconds, and lastly 1 millisecond.

SI #2 uses one sine FMU and five modified integrate FMUs with the same simulation setup as SI #1.

SI #3 uses one sine FMU and 100 integrate FMUs.

#### B. Test Results

This section contains the results of the tests described in Section III-A. The results are presented in tables, where the unit of the numbers is milliseconds, and the table columns represent the following: Sequential refers to the sequential implementation, “Future” refers to the concurrent implementation



TABLE III  
RESULTS FROM SI #2

Wait	Sequential	Future	Par	Actor
0.0	355	434	834	622
0.5	21904	4679	5042	4746
1.0	43356	8970	9348	9184

TABLE IV  
RESULTS FROM SI #3

Sequential	Future	Par	Actor
1464	1432	1967	1857

using futures, “Par” refers to the concurrent implementation using parallel collections, and “Actor” refers to the concurrent implementation using actors. The result for the HVAC test is presented in Table I, and the results for the SI tests are presented in Table II, III, and IV.

Based on these tests it is possible to draw some conclusions: **Executing simulations concurrently can be faster than executing them sequentially:** The results for HVAC #1, SI #2, and SI #3 show that concurrent execution can be faster than sequential execution.

**Executing simulations sequentially can be faster than executing them concurrently:** The results for HVAC #1, SI #1, SI #2 and SI #3 show, that sequential execution can be faster than concurrent execution. Some of these test results contradict the previous conclusion, and therefore it is necessary to pay attention to the concurrency feature used.

**Trade-off:** An interesting discovery is that parallel collections perform worse than futures and actors. This indicates, that even though parallel collections offer fewer capabilities than the other concurrency features, it does not perform faster.

#### IV. RELATED WORK

In order to make use of the improvements in hardware, it is necessary to improve the software. An adage known by “Wirth’s law” goes: “Software is getting slower more rapidly than hardware becomes faster” [17]<sup>5</sup>. He argues, that methodologies are important in order to take full advantage of the improvements in hardware. Sutter urges application developers to take a hard look at the design of their applications and identify places that could benefit from concurrency [19]. This is necessary to exploit hardware capabilities, as processor manufacturers are turning to multicore processors. Harper et. al. conducted a study on a large-scale Publish/Subscribe bus system, and found an overall performance of 80 percent based on concurrency experiments [20]. Additionally, they surveyed concurrency design patterns with the purpose of helping developers towards the “right” patterns.

As mentioned previously, it is important to reduce communication and synchronization overhead between processes to achieve a fast simulation. Agrawal et. al. have implemented and evaluated three communications primitives for hardware/software co-simulation and found, that a message-queue

based communication backplane is preferable [21]. The other two primitives evaluated were shared memory and file-based sockets. Strategies that address the issue of synchronization are also introduced by Bishop et. al., and these strategies also deal with time management [22]. They conclude that using the design strategies discussed can enable the development of high-performance application-specific co-simulations. Kim et. al. consider synchronization between components simulators as the main reason for poor performance of HW/SW co-simulation [23]. They propose a novel technique based on virtual synchronization, which improves the simulator speed and minimizes the synchronization overhead. Becker et. al. describes an approach, where distributed communicating processes are used for the interaction between software and hardware using Unix interprocess communications mechanisms [24]. The approach does not accurately simulate the relative speeds of the hardware and software components, but the author’s found this to be acceptable in their case.

#### V. CONCLUDING REMARKS

Using FMI it is possible to develop a generalised application capable of performing co-simulation, thereby avoiding the need for tailored solutions developed to support the co-simulation of specific systems. It is desirable to perform a co-simulation as fast as possible, as it can help to verify the behaviour of systems or lead to the discover of undesired behaviour. It was therefore investigated whether concurrency could be used to improve the performance of an application performing co-simulation. In some cases the usage of concurrency resulted in faster co-simulations, whereas in other cases sequential computation offered better performance. Because of this it is reasonable to conclude, that it is necessary to allow for different simulation strategies to achieve the fastest simulation. These strategies should support running simulations sequentially, concurrently, or a mix of these. For example, if an FMU that performs long-lasting computations is to be simulated with three FMUs that performs fast computations, then it could be optimal to run this simulation using two threads as shown in Figure 4.

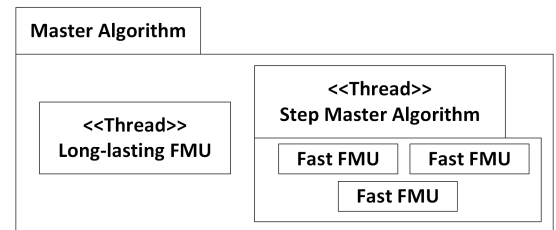


Fig. 4. Master Algorithm simulating four FMUs using an additional step Master Algorithm.

Allowing for different strategies inevitably involves computing which strategies to use. A way of assisting the choice of strategy is to include a measure of how long-lasting the computations performed by an FMU are within the properties of the given FMU. However, this might be difficult to realise

<sup>5</sup>Wirth attributes this to a different saying by Reiser [18].

in a practical manner, where different hardware is used. An alternative approach is to use meta data for a given simulation. This can be configured beforehand, or the COE can determine it, when running the first co-simulation using the given FMUs.

## VI. FUTURE WORK

In order to improve the performance of the COE and choose when to use concurrency, there are several tasks to undertake: **Testability:** Currently, the COE supports reporting the duration of an entire simulation without initialisation and reporting of results. As these steps inevitably are part of a simulation, they should be part of the performance tests. Additionally, the COE should offer better granularity for performance measurements. Better granularity will make it possible to examine the performance of different parts of the application, which can aid in finding bottlenecks and help target the development effort. **Investigate concurrency:** Besides concluding that concurrency can/cannot improve the performance of the application in some cases, it is interesting to investigate when concurrency can improve the performance. Part of this investigation is to determine, whether an increase of performance is achievable by enabling sequential, concurrent, and mixed processing, as mentioned in the previous section. If this investigation results in multiple strategies being implemented in the COE, then it should also be investigated how to configure the COE, so the right strategy for a given simulation is chosen.

**Guidelines:** Since the future work concerns investigation of concurrency, it is compelling to attempt to generalise the lessons that will be learned and apply them on different case studies. The hope is, that this can contribute to existing methodologies and guidelines on using concurrency.

**Semantics alignment:** The continuation of the FMI semantics work referred to above will also involve theorem proving using the Isabelle theorem prover [25] and we hope that it will be possible to align that with the COE work in order to use the semantics directly as an oracle of checking conformance. This also involves examining the semantic properties of the concurrency features.

## ACKNOWLEDGMENT

The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047. Furthermore, the authors would like to thank Nick Battle for reviewing and providing input to this paper.

## REFERENCES

- [1] D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, Sept 2013, pp. 1–12.
- [2] J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *8th International Modelica Conference*, 2011.
- [3] FMI development group, "Functional mock-up interface for model exchange and co-simulation 2.0," Modelica, Tech. Rep. Version 2.0, July 2014.
- [4] J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, "Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains," in *FormaliSE: FME Workshop on Formal Methods in Software Engineering*. Florence, Italy: ICSE 2015, May 2015.
- [5] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.
- [6] Typesafe Inc, "Akka scala documentation," <http://akka.io/docs/>, Akka, September 2015, Release 2.4.0.
- [7] P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises - scala documentation," <http://docs.scala-lang.org/overviews/core/futures.html>, (Visited on 05/03/2016).
- [8] A. Prokopec and H. Miller, "Parallel collections - overview - scala documentation," <http://docs.scala-lang.org/overviews/parallel-collections/overview.html>, 2015, (Visited on 05/03/2015).
- [9] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: <http://doi.acm.org/10.1145/800228.806932>
- [10] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: <http://worrydream.com/refs/Hewitt-ActorModel.pdf>
- [11] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263 – 1277, 1999. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1383762198000678>
- [12] C. T. Hansen, "Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS," Department of Engineering, Aarhus University, Finlandsgade 22, Aarhus N, 8200, Tech. Rep. ECE-TR-26, May 2016. [Online]. Available: <http://ojs.statsbiblioteket.dk/index.php/ece/issue/archive>
- [13] N. Amalio, A. Cavalcanti, C. König, and J. Woodcock, "Foundations for FMI Co-Modelling," INTO-CPS Deliverable, D2.1d, Tech. Rep., December 2015.
- [14] T. Hoare, *Communication Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.
- [15] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3 — A Modern Refinement Checker for CSP," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 8413, 2014, pp. 187–201.
- [16] Microsoft, "Acquiring high-resolution time stamps (windows)," [https://msdn.microsoft.com/en-us/library/dn553408\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx), 2015, (Visited on 05/03/2016).
- [17] N. Wirth, "A plea for lean software," *Computer*, vol. 28, no. 2, pp. 64–68, Feb 1995.
- [18] M. Reiser, *The Oberon System: User Guide and Programmer's Manual*. New York, NY, USA: ACM, 1991.
- [19] H. Sutter, "A fundamental turn toward concurrency in software," *Dr. Dobbs's Journal*, vol. 30, no. 3, pp. 16–23, 2005.
- [20] K. E. Harper, J. Zheng, and S. Mahate, "Experiences in initiating concurrency software research efforts," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 139–148. [Online]. Available: <http://doi.acm.org/10.1145/1810295.1810316>
- [21] B. Agrawal, T. Sherwood, C. Shin, and S. Yoon, "Addressing the challenges of synchronization/communication and debugging support in hardware/software cosimulation," in *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, Jan 2008, pp. 354–361.
- [22] W. Bishop and W. Loucks, "A heterogeneous environment for hardware/software cosimulation," in *Simulation Symposium, 1997. Proceedings., 30th Annual*, Apr 1997, pp. 14–22.
- [23] D. Kim, Y. Yi, and S. Ha, "Trace-driven hw/sw cosimulation using virtual synchronization technique," in *Design Automation Conference, 2005. Proceedings. 42nd*, June 2005, pp. 345–348.
- [24] D. Becker, R. K. Singh, and S. G. Tell, "An engineering environment for hardware/software co-simulation," in *In 29th ACM/IEEE Design Automation Conference*, 1992, pp. 129–134.
- [25] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

# A static approach to estimation of execution time of components in AADL models

Aleksey Troitskiy, Denis Buzdalov  
ISP RAS

**Abstract**—In this paper we work on a problem of estimation of execution time for components appearing in model-based avionics design. We describe one static approach for components of AADL-models with standard behavior specifications based on specialized extended finite-state machines.

## I. INTRODUCTION

Modern avionics is responsible for control of almost all aspects of aircraft operation. As a result, the complexity of such systems is really high. Thus making sure that developed system is correct is a challenging task.

Nowadays problems and their solution bring additional complexity to avionics systems. To satisfy models requirements for weight and power consumption, integrated modular avionics (IMA) approach is used. It means that several resources (e.g. universal processor modules and network) are shared between several pieces of software.

This approach solves weight and power consumption problems, but leads to potential problems of interfering of applications. The approach leads to appearing of step of the integration of the whole system, i.e. deployment of software on different hardware, network configuration and etc. It means that the whole system correctness must be checked and this problem is not solvable by checking of correctness of each part of the system.

The model-driven approach of development allows to manage with the complexity of a system being developed. In particular, models are needed to perform different kinds of analysis of the modelled system though analysis of appropriate models. Such analyses are intended to be performed on different stages of development, in particular, to eliminate errors at early steps of development.

One kind of checks that are needed to be performed is check of *timing properties* of software components.

In particular, during design and deployment stages, each particular application is bound to a processor module. Appropriate timing properties are assigned to them, for example

- *dispatch protocol*, i.e. whether an application is fired periodically, eventually (sporadically) or both;
- *period* of execution for periodic applications;
- *compute deadline*, i.e. time interval in which an application has to finish its work after it was given an ability to execute;
- *recover deadline*, i.e. time interval in which an application has to recover from recoverable errors;

- *data processing time*, i.e. the time between sending an processed output data after getting some input data;
- *output rate*, i.e. rate at which an application has to produce its output, when it is periodic;
- *output jitter*, i.e. maximum deviation of time for periodic output and etc.

Being assigned to some particular application, these properties can be used in schedulability analysis, data flow timing analysis, worst case execution time (WCET) analysis and etc. Some desired or expected values can appear before implementation of particular software.

During the system development, models of it are refined. In particular, for software some behaviour specifications can appear. Such behaviour specifications can be purely functional (i.e. containing only information about which outputs will be produced in particular inputs at the given state).

Also such specifications can contain how much time will be consumed in this or that situation. The addition of this information can lead to inconsistency in the model, because some assumptions about timing properties of software can already exist in the model and these assumptions can contradict with behaviour specification. To check the consistency of a model, it is important to estimate timing properties of particular behaviour specifications.

**Compute deadline consistency example** Consider a periodic software component with some particular *period* set in the model. Consider also that this component has *compute deadline* property bounds set to a range  $p$  from  $p_1$  to  $p_2$  ms. Obviously, higher bound of  $p$  must be not more than period property.

Compute deadline property can be used in the schedule building: e.g. a time frame of  $p_2$  ms can be reserved each period to ensure this software component has enough time to compute. This can be done on early stages of system development when no particular behaviour is known yet.

Consider the case when this software component is refined after some steps of the model development: now its behaviour is specified with automaton with transitions containing how much time is consumed by computations assigned to them. Consider that we have estimated time consumption of the application using this automaton as a range  $h$  from  $h_1$  to  $h_2$  ms each period.

After getting estimations  $h$  we can compare it with bounds  $p$  from the model and there are several decisions we can take:

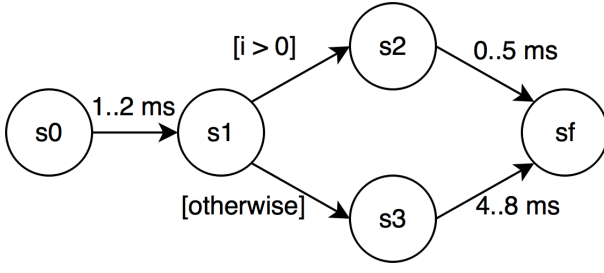


Figure 1. Example of behaviour specification

- when  $h = p$ , behavior corresponds to property and the model is consistent;
- when  $h \not\subset p$ , the model can be inconsistent because real execution time may miss the bounds;
- when  $h \subset p, p \neq h$ , the behaviour specification corresponds to the property; also, we can say that the property in the model can be refined to a more precise value;
- when  $p \cap h = \emptyset$ , the model is inconsistent.

**Example of consistent case** Consider an example when the model has bounds for *compute deadline* property set to be from 3 to 10 ms. Consider also that this application has behaviour specification with automaton shown on the fig. 1. Each period this application begins in state  $s_0$  and finishes in  $s_f$ . In this example we can estimate execution time of the application to be between 5 and 10 ms. This value is consistent with property set in the model.

There is another case when such estimations are useful. Consider a situation when some software component in the model did not have any timing properties set. Consider then, that later it was refined and some behaviour specification has appeared for it. The model still needs to be checked for schedulability and other timing-aware properties. So, we need to derive these timing properties for a component with some behaviour specification. Again, we run into an issue of estimation of timing properties having a particular behaviour specification.

So, generally we can resume that there is an important issue of estimation of timing properties in responsible systems' models with behaviour specifications.

## II. AADL AND BA

We use AADL (Architecture Analysis and Design Language, [1]) as a modelling language. It allows to describe both physical and logical parts of the modelled system, connections between components and bindings between layers of the system. AADL has a mechanism of the language extending though special language annexes and it has a number of standard annexes.

One of such extensions is called Behavior Model Annex [2] (BA). It allows to specify behavior of AADL-components using extended time-aware finite-state machine.

Behaviours are set to components of a modelled system. The basic elements used in BA behaviour specifications are

- automaton states change;
- internal computations;
- accessing and assigning to internal or external variables (data components);
- interaction with the outer world using input/output ports; depending of behaviour, input ports can be managed both by pulling data and by waiting for data to come;
- handling *dispatch* events, i.e. a situation when software component is allowed to perform its execution (e.g., an operating system signals a thread to start).

Behavior Annex automaton must contain a single initial state. When the automaton goes out from the initial state, its internal variables are being initialized. The automaton can contain several final states, in these states automaton can stop its execution.

Each state of the automaton belongs to one of the classes of *complete states* or *execution states*.

Transitions from execution states occur immediately after automaton comes to such state. In complete states automaton waits for external events (data for input ports or dispatch event). Transitions going out of complete states are fired as soon as corresponding event happens.

In BA each state transition is assigned with a list of actions which is run when automaton performs this transition.

There are actions that appear in the list of actions in BA behaviour specification:

- actions with ports: reading, writing, getting of messages count in ports;
- actions with local and accessible external variables: reading and assignment;
- locking on resources: getting and releasing;
- action for modelling of time consumption ( $computation(t_{min}..t_{max})$ );
- **stop** action for automaton interruption;
- composite actions: loops, conditionals;
- computation of arithmetical expressions.

In fact, since loops and conditionals can appear in actions for transitions, every behaviour specification can be represented with complete states only. But using of execution states allows a modeller to express behaviour specification in easier and cleaner way.

## III. PROBLEM

We focus on AADL models with behaviour specifications set using Behavior Model Annex language.

We consider a BA behaviour specification of a single component in a model. Also, we consider two states  $s_{start}$  and  $s_{end}$  of the automaton are given.

We want to estimate the maximum and minimum model time the BA automaton will consume to go out from state  $s_{start}$  and to come to  $s_{end}$ .

#### IV. SOLUTION

Automaton can reach a given state starting from another given state in several ways depending on variables state, external events and nondeterminism. We will call an interleaving sequence of states and transitions as a *path* in automaton.

Thus we divide the original problem to considering a single path in automaton and then considering the automaton itself as a source of paths.

##### A. Path estimations

First, let us look at a finite path starting and ending at given states  $s_{start}$  and  $s_{end}$ , and going through states  $s_1, s_2, \dots, s_n$ , which could be equal to each other and to states  $s_{start}$  and  $s_{end}$ . We would designate it as  $s_{start} \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n \rightarrow s_{end}$ . The question is how long does it takes to go along this path out from  $s_{start}$  to  $s_{end}$ .

Some of states in the path may be complete. An automaton is waiting for external events in these states while going through them. It is a hard task to estimate how much time would it take because it is not a local property, i.e. it depends on other components in the model.

Execution states do not consume any time by definition, thus there is not such problem for them.

Also, in BA actions assigned to transitions can take some time (*e.g. computation action takes time, which is specified with its argument; input/output operations may take time too*). Time taken by composite actions (loops and conditionals) depend on very actions inside them and external conditions (state of variables and ports). Having dependency on external conditions, estimation of time consumption by conditionals it a tricky task (undecidable in the general case).

Thus, task of estimation of time, taking by execution of a finite path, can be split into two tasks: time estimation for each complete state in the path and for each list of actions assigned to a transition in the path.

##### B. Automaton estimations

The whole automaton containing both execution and complete states is a challenging object. Let us at first consider simpler kind of automaton containing only execution states and then to consider the general case.

1) *Automata with execution states only*: In this case automaton is not waiting for external events and goes through states right away.

We can represent such automaton as a weighted graph. Vertexes of the graph are states of the automaton, and edges of the graph are transitions of the automaton. Weight of each edge is time estimation for the actions of corresponding transition.

We can use all known algorithms for finding minimum and maximum times (*e.g. for finding minimum time we can use Dijkstra's algorithm [3]*).

However, when the graph is cyclic these estimations can be inaccurate. *For example, we have a loop of the*

*automaton which is executed exactly 50 times. If this fact is not used, estimation of the time consuming by this loop may be too imprecise, up to  $+\infty$  for the higher bound and to 0 for the lower bound. Considering information of the number of loop iterations, we can estimate the time to be  $50 \cdot t_{body}$  where  $t_{body}$  is an estimation of the time consuming by the loop body, or even more precise if  $t_{body}$  depends on the loop iteration number in a known way.*

Despite inaccuracy in some cases, time estimation for this kind of automaton is a pretty studied problem.

2) *Automata with complete states too*: Approaches with simple Weighted graphs with weights only on edges do not model the fact that automaton can wait some time in a complete state during its execution. But we work with automata having complete states. Thus we need to manage with it somehow while estimating automata execution time.

It seems that this problem can be reduced to the previous one, e.g. though replacing a single complete state with two connected execution states with a transition consuming the same time as automaton waits in this complete state.

But what we realized trying to implement such approach is that time of waiting in a complete state is not local and cannot be represented by some constant. This time actually depends both on the way this state was reached and on how regular external events occur. So, automata with complete states need special treatment, one variant of which will be discussed below.

##### C. Resume

So, to solve the original task we have divided the original problem to the following subtasks:

- estimation of time consumption of paths in automaton:
  - estimation of execution time for transitions;
  - estimation of time of waiting in complete states;
- estimation of time consumption by automaton itself:
  - in a particular case, when the automaton contains only execution states;
  - in the general case, when automata with both complete and execution states are considered.

The rest of the paper follows this division.

#### V. ESTIMATION OF TIME FOR PATHS

##### A. Estimation of time for transitions

Let us estimate how much time can take different Behavior Annex actions. At first, look at simple actions.

The action `computation` has a time as an argument, which is the execution time of this action.

Also the action `get resource` can take some time, because at the moment when this action is executed, needed resource can be used by some other component. And so it will be necessary to wait for some time until the resource can be used. We will estimate this time from 0 to  $+\infty$ .



If action **stop** occurs at some point, then the execution of automaton became interrupted and it does not go to the next state. The action does not take time. However, since we are interested in the time between the states of the automaton, it is convenient to assume that the time of this action is  $+\infty$ . Indeed, if the transition from  $s$  to  $q$  with action **stop** exists, it means that automaton will not ever be in state  $q$  after this transition.

Now let us consider composite actions. Loops, which contains the actions occupied some time, we will estimate with time from 0 to  $+\infty$ . Other loops do not take any time.

We will estimate conditional constructions with time the from 0 to the maximal time, which could take the actions performed when the condition is true. In this way, estimations for transitions of the automaton can be performed.

Now let us estimate time, which automaton is waiting in complete states.

### B. Estimation of time for complete states

Behavior Annex allows to handle two types of external events: receiving message to input port and dispatch signal.

At first, look at the first type of events. Since the expectation of the receiving message can take arbitrary large time, we will estimate this time with 0 to  $+\infty$ . So, this is the estimations of time of waiting in the complete states for the external event of the first type.

Estimations of time waiting for events of the second type can be performed more accurately when the component is a thread. This is due to the fact, that AADL allows to set properties for the thread, which determined how often dispatch signal arrives to the thread (*this properties are Dispatch Protocol and Period*).

So, this properties determine the time between neighboring complete states in automaton. Consider any path from the graph, which starts and ends in complete states, all other states are execution, and the transition from the first complete state is the transition of the second type. Above AADL-properties can determine the execution time of this path from going out from the first complete state to going out from the second complete state. This time can be determined hard, or it can have only lower or upper bound, or it can be not limited.

In this way, when automaton comes to complete state, the waiting time in this state is determined by the time elapsed from going out from the previous complete state and by the AADL-properties.

## VI. ESTIMATION OF TIME FOR THE WHOLE AUTOMATON

### A. Particular case, execution states only

1) *Problem:* The weighted oriented graph  $G = \{V, E\}$  and two vertices  $s_{start}, s_{end}$  are given. The weights of the edges are determined by the function  $f : E \rightarrow \mathbb{R}^2$ . Weight of each edge is a range of two real numbers  $[r_1, r_2]; r_2 \geq r_1$ ,

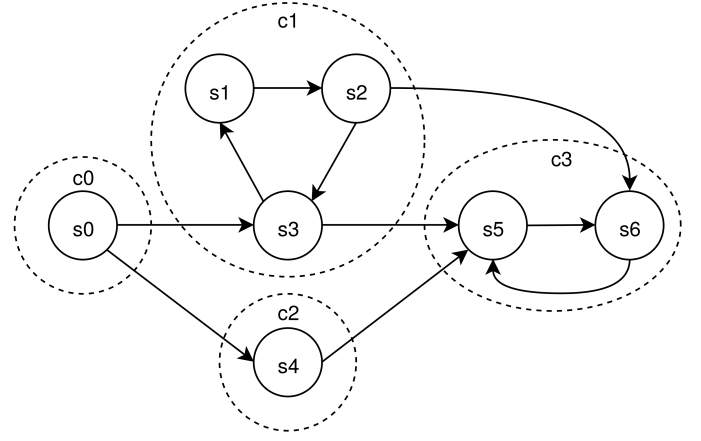


Figure 2. Graph G and strongly connected components

$r_1$  — lower bound,  $r_2$  — upper bound of the range. The set of weights is partially ordered by the native function:

$$[r_1, r_2] < [q_1, q_2] \Leftrightarrow r_2 < q_1$$

Also the adding function for weights is determined:

$$[r_1, r_2] + [q_1, q_2] = [r_1 + q_1, r_2 + q_2]$$

The problem is to find the weights of the longest and the shortest paths from  $s_{start}$  to  $s_{end}$ .

For example, we will consider the graph on the fig. 2 and vertices  $s_0$  and  $s_6$  as  $s_{start}$  and  $s_{end}$  respectively.

#### 2) Algorithm:

- 1) We find strongly connected components (SCC) in graph  $G$  with Tarjan's algorithm [4]. Strongly connected components of the graph  $G$  are highlighted by a dotted line on fig. 2.
- 2) We build acyclic graph  $E$  from strongly connected components of the graph  $G$  (fig. 3).
- 3) Let vertices  $s_{start}$  and  $s_{end}$  belong to strongly connected components  $c_{start}$  and  $c_{end}$  respectively. Then we find all paths in acyclic graph  $E$  from  $c_{start}$  to  $c_{end}$ . In the example, this is all paths from  $c_0$  to  $c_3$ :  $c_0 \rightarrow c_1 \rightarrow c_3$  and  $c_0 \rightarrow c_2 \rightarrow c_3$ .
- 4) For each SCC-path  $c_{start} \rightarrow c_1 \rightarrow \dots \rightarrow c_{n-1} \rightarrow c_{end}$  we find paths from  $s_{start}$  to  $s_{end}$  as:

$$s_{start} \rightsquigarrow s_0^{out} \rightarrow s_1^{in} \rightsquigarrow s_1^{out} \rightarrow s_2^{in} \rightsquigarrow \dots \rightsquigarrow s_{n-1}^{in} \rightsquigarrow s_{n-1}^{out} \rightarrow s_n^{in} \rightsquigarrow s_{end} \quad (1)$$

There  $s_i^{in}, s_i^{out} \in c_i; i \in [1..n-1] \cup \{start, end\}$ , and edges  $e_j = (s_j^{out} \rightarrow s_{j+1}^{in}), e_j \in E, i \in [1..n-1] \cup \{start, end\}$ . The arrow  $s_i^{in} \rightsquigarrow s_j^{out}$  represents automaton walking in one SCC-component from state  $s_i$  to state  $s_j$ .

And besides vertices  $s_i^{in}$  and  $s_i^{out}$  could be coincided. In this case, if SCC-component  $c_i$  contains only one state, which does not have an edge to self (for example, SCC-component  $c_2$  from the example), then the transition  $s_i^{in} \rightsquigarrow s_i^{out}$  has zero weight and it can be removed from the path.

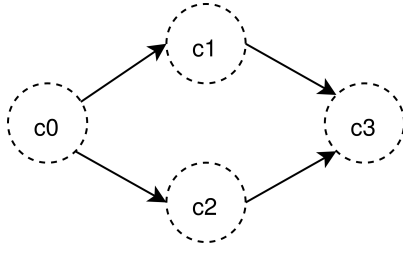


Figure 3. Graph E.

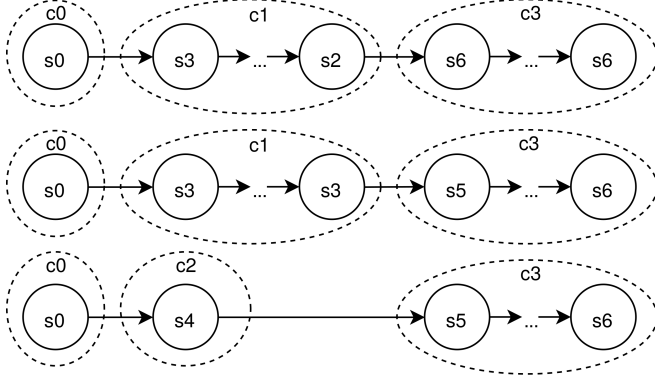


Figure 4. Paths in graph G from s0 to s6.

On the fig. 4 all paths are presented.

- 5) Let find the weight of each path like (1). To do that we need to estimate transitions:  $s_i^{in} \rightsquigarrow s_i^{out}, i \in [1..n-1] \cup \{start, end\}$ , as weights of other transitions determined by the function  $f$ .

So, now we need to solve our problem for strongly connected component.

To find the weight of the shortest path we can use one of well-known algorithms (e.g. *Dijkstra's algorithm* [3]). To do that we need to replace weights of the edges:  $[r_1, r_2] \rightarrow r_1$ .

The weight of the longest path in SCC-component is  $+\infty$ , if this SCC-component is cyclical (contains more than one vertex, or has the only vertex with edge to itself).

#### B. General case, both execution and complete states

1) *Problem:* The Behavior Annex automaton and two states of the automaton are given. The problem is to find estimations for the execution time of the automaton from exit from the state  $s_{start}$  to enter to the state  $s_{end}$ .

The set of states of the automaton defined as  $S$ , the set of transitions of the automaton defined as  $T$ . The set of execution states of the automaton is  $Exec \subset S$ , the set of complete states of the automaton is  $Comp \subset S$ .

For example, let consider the automaton on fig. 5. Complete states are marked by white color, execute states are gray. The goal is to find time between state e2 and state c2.

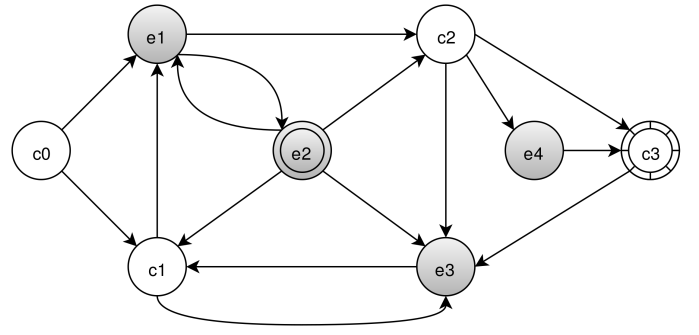


Figure 5. Graph with complete states and execution states.

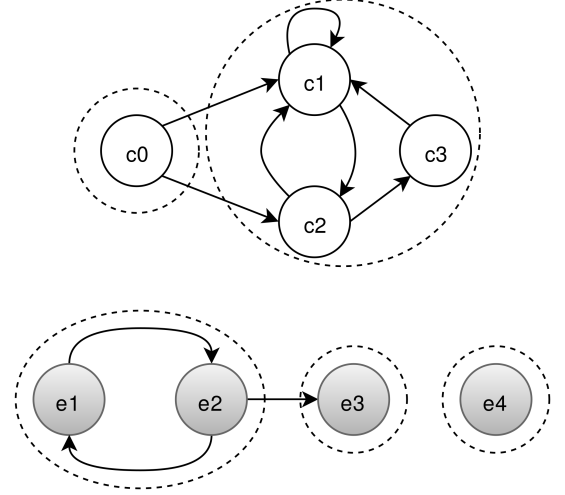


Figure 6. Graph  $G_e$  and graph  $G_c$ .

2) *Solution idea:* Two different states types are determined in Behavior Annex. So we consider two different graphs.

We consider graph of the complete states and the graph of the execution states separately. Then if we need to find time between exit from one complete state to exit from other complete state, we use graph of complete states. In other cases we use the graph of execution states.

3) *Algorithm:*

- 1) Let define weighted oriented graph  $G_e$ . The vertices of the graph  $G_e$  are all execution states of the automaton. For each transition  $e_1 \rightarrow e_2$  of the automaton we build edge  $e_1 \rightarrow e_2$  in graph  $G_e$ . The weight of this edge is time estimation of transition's actions (sec. V-A).

Graph  $G_e$  could be not connected.

Graph  $G_e$  is presented on fig. 6 on the right.

- 2) We build weighted oriented graph  $G_c$ . The vertices of the graph  $G_c$  are complete states of the automaton. We will build edge  $c_1 \rightarrow c_2$ , if a path like  $c_1 \rightarrow e_1 \rightarrow e_2 \dots \rightarrow e_n \rightarrow c_2$ , where  $e_i \in Exec; n \in [0, +\infty]$ , exists in automaton. This path does not come through any of complete state besides  $c_1$  and  $c_2$ .

In particular case, if the automaton contains transition from  $c_1$  to  $c_2$ , then graph  $G_c$  will contain edge



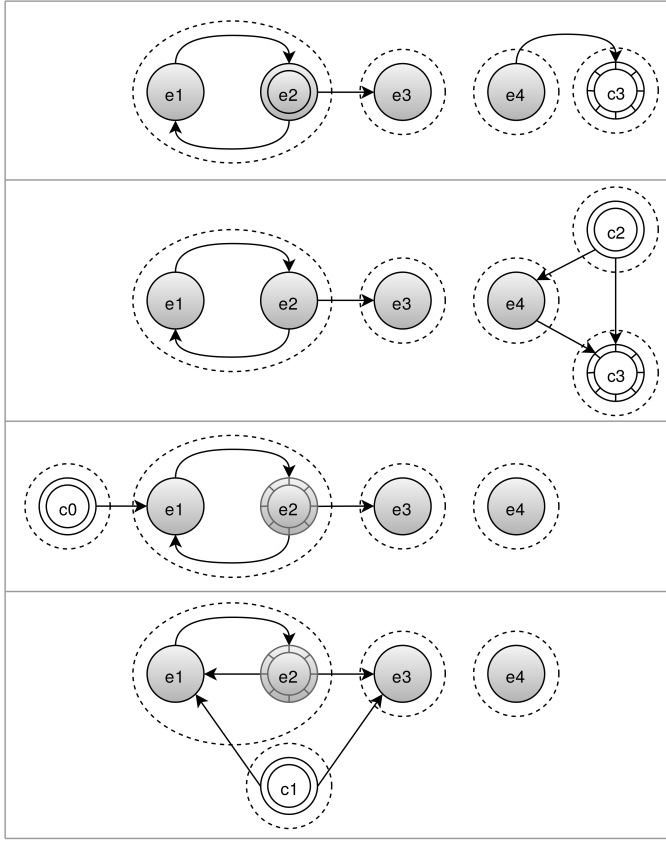


Figure 7. Usage of graph  $G_e$ .

from  $c_1$  to  $c_2$ . Weights of edges are determined with AADL-properties of the component as it described in the section V-B.

Graph  $G_c$  for the example is presented on fig. 6.

- 3) Graphs  $G_e, G_c$  and the algorithm described in the previous section we will use as follows.

When we need to find time between exit from the complete state to enter to complete state, we will execute an algorithm on graph  $G_c$ .

With graph  $G_e$  we can find time between exit from state  $q \in S$  to enter to state  $p \in S$  across only execution states. If state  $q$  is complete state, we will temporarily push it to graph  $G_e$  (fig. 7). If state  $p$  is also complete, we will temporarily push it too.

Also, we will temporary push edges, which are corresponding to out transitions (from  $T$ ) from state  $q$  to states from  $G_e$  and in the transitions from states to  $p$ . After that we can execute the algorithm on updated graph  $G_e$ .

On the second line of fig. 7 the graph  $G_e$  for calculating the time between exit from complete state  $c_2$  to enter to complete state  $c_3$  is presented.

- 4) Let consider two sets:  $PREV_{out}$  and  $PREV_{in}$ . If state  $s_{start}$  is complete, we define as follows:  $PREV_{out} = \{s_{start}\}$ . If state  $s_{start}$  is execution, we find all previous complete states for state  $s_{start}$  (e.g. with DFS). It is possible to go state  $s_{start}$  from any of that complete states. Similarly,  $PREV_{in}$  is set of possible previous complete

states for state  $s_{end}$ .

In example for  $e_2$  and  $c_3$ :  $S_{out} = \{c_0, c_1\}$ ,  $S_{in} = c_2$ .

- 5) Let introduce set  $Times = \{\}$ , initially it will be empty. We find the length of the way from exit from state  $s_{start}$  to enter to  $s_{end}$  with graph  $G_e$  (fig. 7 first line for the example). If this way exists and the estimation does not equal to  $+\infty$ , we add the estimation to set  $Times$  (on the example it does not exist).
- 6) For each  $c_{out} \in PREV_{out}$  and  $c_{in} \in PREV_{in}$  we find estimations for paths:  $t_e(c_{out} \rightarrow s_{start})$ ,  $t_e(c_{out} \rightarrow c_{in})$ ,  $t_e(c_{in}, s_{end})$  and add to  $Times$  estimation

$$t_c(c_{out} \rightarrow c_{in}) - t_e(c_{out} \rightarrow s_{start}) + t_e(c_{out} \rightarrow s_{end})$$

- 7) The result of the algorithm is the minimum time range, that contains all time ranges from the set  $Times$ .

## VII. RELATED WORKS

One close problem to the problems, considered in this paper, is WCET problem. This problem is famous, and a lot of algorithms looking for WCET exist. But these algorithms cannot be applied to our problem directly, due to considered specific object class, defined by Behaviour Annex language. As Behavior Annex describes behaviour based on timed automata, consider WCET algorithms working on timed automata.

The WCET problem for timed automata was considered in the paper [5]. This paper has a description of the algorithm using the difference-bound matrix data structure to represent zones (heuristic). This algorithm can be applied in the particular case, which was described upper, in the following way.

The main specific construct in Behaviour Annex is complete states. In the particular case we consider automata with only execution states. These automata are very similar to timed automata from the paper [5]. It means that algorithms from the paper can be applied to the particular case. We are thinking out about applying it, but currently we have chosen simpler algorithm.

But to use it in the general case, it should be adapted. We have decided that the adaptation of the algorithm would be harder, than to develop the new algorithm applied to a needed object class.

## VIII. RESULTS AND FUTURE WORKS

The algorithm for finding time estimations of execution time of behaviour of AADL-components on Behavior Annex was developed. It was realized in tool MASIW [6] — IDE for development and analysis of AADL models.

Characteristics of behaviours, that are got with algorithm, could be used for checking model consistency and for model refinement, when AADL-properties have not given.

## REFERENCES

- [1] *Architecture Analysis & Design Language (AADL)*, SAE International standard AS5506B, SAE International, 2012, <http://standards.sae.org/as5506b/>.
- [2] *Architecture Analysis & Design Language (AADL), Annex Volume 2, Behavior Model Annex*, SAE International, 2011, <http://standards.sae.org/as5506/2/>.
- [3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1959.
- [4] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, 1972.
- [5] O. I. Al-Bataineh, "Verifying worst-case execution time of timed automata models with cyclic behaviour," Ph.D. dissertation, School of Computer Science & Software Engineering, 2015.
- [6] D. Buzdalov, S. Zelenov, E. Kornychin, A. Petrenko, A. Strakh, A. Ugnenko, and A. Khoroshilov, "Tools for system design of integrated modular avionics," in *Proceedings of the Institute for System Programming of RAS*, vol. 26, no. 1, 2014, pp. 201–230.

# *Practical experience of software and system engineering approaches in requirements management for software development in aviation industry*

*Igor Koverninskiy, Anna Kan, Vladimir Volkov, Yuri Popov, Natalia Gorelits*  
Department 2100  
State Research Institute of Aviation Systems (GosNIIAS)  
7 Viktorenko Str., Moscow, 125319, Russia  
nkgorelits@2100.gosniias.ru

**The article describes the technical world evolution tendencies, which require software and system engineering approaches used for complex systems creation. Basics of software and system engineering are set out. Information systems which have been created in GosNIIAS are considered: information system of requirements management, information system of problem reports management, technological environment for test methods preparation and test results registration. Some perspective directions of software and system engineering approaches applying in GosNIIAS are listed.**

*Software engineering; system engineering; requirements management; complex on-board equipment; aircraft design*

## **I. INTRODUCTION**

Nowadays there is a considerable change in industries all over the worlds. The change is related with the rapidly increasing complexity level of systems and devices which are created and used.

Safety and reliability requirements to products of aerospace, defense and other industries become stricter as well as certification requirements to management processes of products creation. At the same time we have to use new industry standards.

Aerospace imposes some restrictions and requirements on the software development process and its result. These restrictions are caused by safety requirements to the aircrafts on which the software will be used. Requirements are set out in the industry standards, these standards must be complied very carefully for high quality results and successful certification.

## **II. SOFTWARE AND SYSTEM ENGINEERING APPROACHES REALIZATION**

Using and customizing software and system engineering processes and approaches are an appropriate response to technical world complication tendencies. These processes and approaches are base of the most standards and guidelines which define methods to achieve necessary safety and reliability levels during development, design and engineering of critical technical and software systems.

Nowadays software in complex technical systems is responsible for executing of the most critical functions [1].

The most important discipline of software and system engineering for software development is requirements management. If there is no requirement management process or its bad realization then obvious or hidden defects and faults appear. It takes more and more efforts to repair these defects and faults at the later stages of development lifecycle.

Problems in requirements are leaders in projects failures reasons lists and rework costs lists (Standish Group reports).

That's why requirements are mandatory basis of design and development processes according to guidelines of standards R4754 (R4754A is now a draft, it is Russian analogue of ARP 4754), KT-178 (DO-178), KT-254 (DO-254), DO-330, GOST R 51904. Development of the software, hardware and systems begins from creation of requirements. Design is based on requirements. We also have to inspect how result corresponds with initial requirements during verification, validation, testing processes.

Some important tasks arose GosNIIAS due to the changes in the world. These tasks were about modernization of existing approaches and work processes in order to minimize potential risks for software design and development [2].

A number of current situation researches were done in GosNIIAS. Existing world approaches to the software and system engineering approaches were adapted considering the specialization of the institute. The results of analysis and adaptation as well as software and system engineering fundamental principles formed the basis of newest works of GosNIIAS.

Fundamentals of software and system engineering:

- Requirements are base of software development process,
- There should be coherent architecture of modules/subsystems and communication interfaces (points of input and output) between modules should be predefined,

- Verification process (product check for requirements compliance) should be organized for cases when accurate measurement is impossible,
- Modeling approaches and then model verification and validation are used for earlier failures and bug detection,
- Communication protocols between process participants should be defined like strict regulations.

Nowadays GosNIIAS has built the number of systems accordingly to software and system engineering approaches. The list of created systems consists of the following systems:

- Requirements management information system,
- Problem reports management information system,
- Technological testing environment,
- Practical approaches and skills in software and system engineering adapted for real tasks.

#### *A. Requirements management information system*

Requirements management information system (RMIS) was created for support requirements management activities in design and development of complex systems like aircraft onboard software.

RMIS processes are built based on R4754 (ARP 4754) processes.

RMIS realizes such functions and processes like:

- Cross-cutting requirement management process during the software and system development entire lifecycle,
- Single requirements change and configuration management process,
- All necessary lifecycle artifacts tracing,
- Generation and publishing of reporting documents and documents with any necessary data in accepted formats.

Documents and projects templates required by standards R4754, KT-178, KT-254, DO-330, GOST R 51904, GOST 34 are created and included in RMIS suite. These items allow to decrease labor costs for audit preparation and passage in certification authorities – processes and products must strictly comply the standards.

Some methodological materials were made to help with requirements management and configuration management using RMIS.

Using RMIS while designing and developing aircrafts allows to significantly reduce:

- Efforts for execution of works,
- Time for approval, negotiation and final products release,
- Errors from difficult work with requirements,

- Provides actual information to all the participants during entire development lifecycle.

This way RMIS gives opportunities to make reasonable and timely decisions.

RMIS was successfully implemented in some organizations. The list of successful users of RMIS in aviation industry includes companies such as GosNIIAS, SpecTechnica, Techodinamika and others.

GosNIIAS effectively uses RMIS in testing avionics processes on integration stand for Irkut MS-21 aircraft. RMIS's database contains traced data from AP-25 (like EASA CS-25, FAR-25 – Airworthiness standards for transport categories airplanes), Certification basis, Special technical conditions and some other data for Irkut MS-21 aircraft. There is active ongoing process of creation, customization and implementation of requirements management process, configuration management process, verification and validation management process in GosNIIAS.

#### *B. Problem reports management information system*

Specialists from GosNIIAS also made Problem reports management information system (PRMIS) during MS-21 project. PRMIS allows support of problem reports management activities on testing avionics processes on integration stand for MS-21 aircraft.

PRMIS processes are built on the base of R4754A (R4754A's part about problem reports activities). Main of PRMIS tasks are

- Collection and storage data of problem situations,
- Problem analysis,
- Resolving problem documenting,
- other functions.

#### *C. Technological environment for test methods preparation and test results registration*

Technological environment for test methods preparation and test results registration (TET) was made during MS-21 project as well. TET allows support of test methods preparation and testing activities on integration stand for MS-21 aircraft's avionics testing. Processes of TET are built in accordance with industry standard R4754.

TET provides the following functions:

- Preparation of test programs, test methods, test cases and test procedures for avionics, integrated flight control system testing,
- Maintenance of testing activities on integration stand,
- Creating test reports,
- Other functions.

TET provides such opportunities as:

- Test methods approval processes,

- Test methods development history logging,
- Test results control and changing of succeeding test methods accordingly to revealed remarks for test requirements, hardware, methods, etc.

Some of TET goals are:

- Reducing labor costs for test methods, test procedures and test cases creation,
- Transparent control for finished tests considering received and registered test results,
- Increasing quality of tests traced with requirements, test methods and programs and received results,
- Possibility to work with the set of integrated hardware on the integration stand,
- Information integration with RMIS, PRMIS and configuration control system for further integration in entire software and system engineering process of GosNIIAS, which will allow effective reusing of prepared test organization process for certification audit.

### III. CURRENT AND FUTURE TASKS

Nowadays there are actively realized system engineering approaches in GosNIIAS. Some tasks about development, design and implementation such processes of system engineering as requirement management process, problem reports management process, information management process,

verification and validation management process, version and configuration management processes during software and system development lifecycle processes.

Processes listed above and traced with its software and system engineering approaches will be performed for the further researches. Real-time operation system creation and creation of Russian instrumental set for support of the software and system engineering processes were chosen as nearest researches for perform these processes. There were defined some models for chosen researches – change request lifecycle processes model and problem report lifecycle processes model.

### IV. CONCLUSION

GosNIIAS has plans to create cross-cutting process based on developed processes and realized with software which is already developed and which will be developed soon. It should be cross-cutting process of software and system engineering with necessary instrumental support in GosNIIAS.

### REFERENCES

- [1] G.A. Chuyanov, V.V. Kosyanchuk, N.I. Selvesyuk, "Prospects of development of complex onboard equipment on the basis of integrated modular avionics," in *Izvestiya SFedU*, vol. 3, pp. 55-62, March 2013 (in Russian).
- [2] G.A. Chuyanov, V.V. Kosyanchuk, N.I. Selvesyuk and S.V. Kravchenko, "Directions of perfection on-board equipment to improve aircraft safety," in *Izvestiya SFedU*, vol. 6, pp. 219-229, June 2014 (in Russian).

# Design and architecture of real-time operating system

Kurbanmagomed Mallachiev

Institute for System Programming  
of the Russian Academy of Sciences,  
CMC MSU,  
Moscow, Russian Federation  
[mallachiev@ispras.ru](mailto:mallachiev@ispras.ru)

Nikolay Pakulin

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
[npak@ispras.ru](mailto:npak@ispras.ru)

Alexey Khoroshilov

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
[khoroshilov@ispras.ru](mailto:khoroshilov@ispras.ru)

**Abstract**—The Integrated modular avionics (IMA) architecture describes real-time computer network airborne systems. ARINC 653 is a specification for software partitioning constrains to the underlying safety-critical avionics real-time operating system and for associated application programming interfaces.

Most existing partition based operating systems with ARINC 653 support are commercial and proprietary software.

In this paper, we present Jet OS, an open source real-time operating system with ARINC 653 support with time and space partitioning, inter- and intra-partition scheduling and complete implementation of ARINC 653 part 1 rev 3 API

**Keywords**—ARINC 653; RTOS; IMA; partitioning; real-time.

## I. INTRODUCTION

Real-time Safety-critical systems have strong requirements in terms of time and resource consumption. Most of them have several concurrently executing separate functions (applications), which communicate from time to time. The most obvious approach is running those applications on separate devices and connecting to sensors and actuators by point-to-point link, on which applications should communicate. But firstly, there will be a lot of wires in large system. And secondly, having a separate computing node for periodic application, which is idle most of the time, results in a great number of computing nodes and high cost of hardware.

Integrated modular avionics (IMA) network is a solution to those problems in avionics. Core modules are main part of IMA network. Core module runs a real-time operating system (RTOS), which supports independent execution of several avionics applications that might be supplied by different vendors. System provides partitioning, i.e., space and time separation of applications for fault tolerance (fault of one application doesn't affect others), reliability and deterministic behavior. The

unit of partitioning is called *partition*. Basically partition is the same as process in commodity operating systems. ARINC 653 standardizes constraints to the underlying RTOS and associated API. [1]

Civil aircraft airborne computers are mostly PowerPC architecture. In this paper we present the project on development of an open source ARINC 653 compatible operating system, which can run on PowerPC CPU and, in the future, on other CPU architectures, such as MIPS and x86.

## A. Overview of ARINC 653

ARINC 653 is the standard for implementing IMA architecture, it defines general purpose APplication Executive (APEX) interface between avionics software and underlying real-time operating system, including interfaces to control the scheduling, communication, concurrency execution and status information of its internal processing

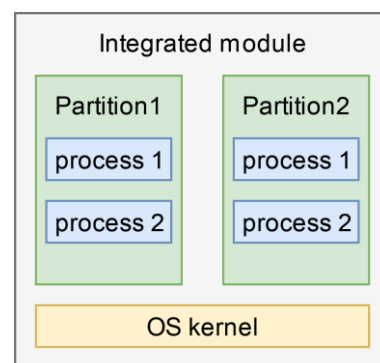


Fig. 1. Example module architecture

elements.

Key concept of ARINC 653 is partitioning of applications in integrated module by space and time. [2]. A partition is a partitioning program unit representing an application. Every partition has its own memory space, so one partition cannot get

access to the memory of another. Partitions are executed in user (non-privileged) mode, so errors in partition cannot affect OS kernel (which is executed in privileged mode) and other partitions. Partition consists of one or more processes, which operate concurrently. Processes in partition have the same address space and can have a different priority. Process has an execution context (processor registers and data and stack areas), and they resemble well-known concept of threads. Fig 1 shows example architecture.

Partitions are scheduled using a simple round-robin algorithm. System defines a major time frame of fixed duration which is constantly repeated through integrated module execution time. Major frame is divided into several time windows. Each partition is assigned to one or more time windows, and partitions are running only during corresponding assigned time window. Assignment of time windows and major frame duration are statically configured by the system integrator, therefore scheduling is fully deterministic.

Scheduling of processes within partition is a dynamic priority based scheduling and communication and synchronization mechanism make it more sophisticated than partitions scheduling.

ARINC 653 provides interface for communication between applications (partitions), potentially running on different modules connected by onboard communication network. All inter-partition communication is conducted via messages. Message is a continuous block of data. The ARINC 653 interface doesn't support fragmented messages. Message source and destination are linked by channels; a channel links a single source to one or more destinations. Partitions have access to channels via defined access points called ports. Port has single direction; it can be either source or destination port. One port can be assigned only to one partition. Each partition can have multiple ports. It is even possible to have a channel where both source and destination ports are assigned to one partition

Partition code works with ports regardless of underlying channels. Channels are preconfigured statically.

To control the concurrent execution of processes ARINC 653 offers synchronization primitives such

as semaphores, events and mutexes. Buffers and blackboards provide inter-process communication within a partition. Buffer is a messages queue, while blackboard has only one message, which is rewritten by every write operation.

## II. RELATED WORKS

ARINC-653 requirements results in constraints to underlying operating system. OS must support:

- space partitioning, so partitions have no access to memory areas of the other partitions and OS kernel;
- time partitioning, so not more than one partition can run at any time;
- strict and determinate inter-partition scheduler that ensures application response time.

Furthermore in safety-critical systems the operating system must undergo certification process. As a result, size and complexity of OS become a real issue.

Popular real-time operating systems (such as RTERMS [3] and FreeRTOS[4]) don't support ARINC 653. Furthermore RTERMS doesn't support memory protection.

Operating systems that satisfy all of these constraints exist, but they are commercial and proprietary software. They are VxWorks[5] (by Wind River), PikeOS[6] (by Sysgo), LynxOS [7](by LynuxWorks).

There are research projects on real-time and ARINC 653 [12] enhancements of Linux. But Linux is a large system, so certification of Linux kernel seems impossible.

There are research projects that exploit the virtualization technology to support ARINC 653. But they are either proprietary like LithOS[8] (works over open hypervisor XtratuM[9]), or limited prototype link VanderLeest implementation of ARINC 653 over Xen [10].

Only POK operating system [13], which is available under BSD license terms, mostly satisfies our requirements, so we decided to fork POK and continue its development.



### III. POK

POK is a partitioned operating system focused on safety and security [11]. We describe it in detail here since it is the basis for the Jet OS that we are working on.

POK has been designed for x86 and ported to PowerPC (PReP) and Sparc. POK has two layers: kernel and partition, where services of partition layer run at low-privileged level (user mode), and kernel services are executed at high-privileged level (kernel mode). Besides the kernel POK provides a library for partition code (libpok), which translates ARINC 653 API to POK kernel syscalls. Fig 2 shows POK architecture.

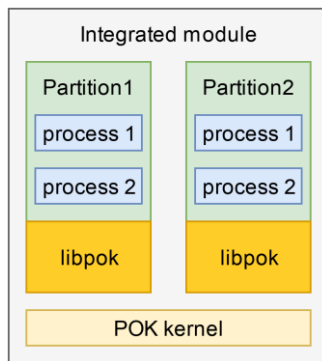


Fig. 2. POK architecture

We selected POK as the basis for our RTOS. Below in this paper we describe parts of POK that were changed or rewritten. We describe limitations of current implementation or architecture of these parts.

**Partition management.** POK provides partition isolation:

- in time by allocating fixed time slots for partitions in the schedule,
- in space by associating a unique memory segment to each partition.

Partition scheduling and memory management of POK partly comply the ARINC 653 specification. But PowerPC processor, on which we focus (P3041), doesn't support memory segmentation.

**Processes management.** POK supports ARINC 653 partition processes. All processes are represented in the kernel as array entries of a single processes array that stores process information for

all partitions. POK has no logical separation in kernel representation of ARINC-653 processes of different partitions.

POK supports two intra-partition schedulers: Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF). Those partitions schedule processes within a partition when its time slot is active.

The problem with POK scheduler is that ARINC 653 requires much more from intra-partition scheduler: priority scheduling and fault management.

POK runs both inter- and intra-partition schedulers in the kernel mode.

**Inter-partition communication.** For every ARINC port there is a buffer of corresponding size inside the kernel. User code while sending to (or receiving from) port accesses those buffers by means of syscalls. At the beginning of every major frame POK copies data from source buffers to destinations. For large buffers there is possibility to spend significant part of partition time slot on buffer-to-buffer copying.

If a process tries to send to a full port (or read from an empty one) the kernel blocks the process until buffer becomes operational. POK supported this feature but did not obey to the ARINC-653 requirement on that the order of unblocking should be the same as the order of blocking on each priority level.

**Intra-partition communication** support is implemented by the user-mode library libpok, using system calls for synchronization purpose. It supports locking resources for concurrent access to shared data resources (such as buffer and blackboards) between processes in partition. When process tries to accesses a locked resource, it will be blocked (so scheduler will skip this process) until the resource is unlocked.

POK scheduler has some inherent problems with handling of locked processes. Let's consider an example. A low-priority locks a buffer for writing and before it unlocks the buffer a higher priority process wakes up. POK scheduler unconditionally switches to the second process. If the second process tries to get status information about the locked buffer it blocks and POK wakes the first process.

But according to ARINC-653 standard the process that requests status information must not block.

#### IV. JET OS

Jet OS is the real time operating system with ARINC-653 support that we currently develop at ISPRAS. It originates from POK but has evolved significantly since then.

Before we introduce the new features of Jet OS compared to POK let us mention the facility that was removed from POK: the AADL configuration tool. Originally POK was designed and implemented as a demonstration of a number of approaches, and the developed selected rather exotic approach to configuration. The suggested way to create an embedded application by means of POK is to specify its environment and capabilities as an AADL specification. In Jet OS we dropped AADL support in favor of XML-based configuration files.

Furthermore we dropped support of the SPARC platform as there are no onboard avionics systems that are built atop of SPARC CPUs. At the moment Jet OS runs on x86 and PowerPC (Book E branch).

**Partition management.** Unlike x86 and SPARC the new target hardware for Jet OS, PowerPC platform, features direct MMU control through TLB writes. To reduce cache flushes at context switches and simplify TLB lookups PowerPC provides tagged cache where each tag is an 8 bit identifier. We use that identifier as partition identifier (pid). At context switch we just change value of the special-purpose register responsible for current pid. This is simple and secure method.

The inter-partition scheduler of POK was able to switch partitions only when the active process runs in user mode. If a process calls syscall it cannot be switched until the end of that call. Such behavior violates requirements of real-time since system calls might be prolonged. Currently we are working on kernel-mode critical section and synchronization primitives to enable context switch while a process executes a system call.

**Processes management.** We store process-related data in kernel separately for different partitions. Intra-partition scheduler was fully rewritten to support ARINC 653 specification. The new scheduling facility allows for multiple scheduler, and different partitions might utilize

different schedulers (a.g. ARINC-653 for avionics applications and preemptive pthreads for system partitions). New intra-partition scheduler can be accessed only by functions

- start() is called when partition is starting or restarting
- on\_event() is called on every event such as timer interrupt and returning control to partition.

**Inter-partition communication.** We use one ring buffer for every channel. Its size is the sum of source and destination ports buffers size in original POK design. It removes the need for copying from source to destinations buffers. Correct work of send and receive function achieved by two pointers, one for source port, and one for destination. Sending increases source port pointer, receiving increases destination port pointer. When pointers are met then buffer either full or empty, uncertainty is resolved by another variable associated with the channel, which stores current number of messages in the channel's buffer. Example can be seen at Fig 3.

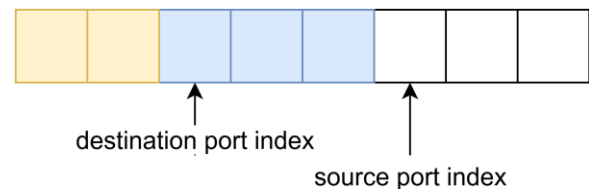


Fig. 3. Example kernel channel buffer. Yellow cells are already received messages, blue cells are sent but not yet received messages, white cells are empty

**Intra-partition communication.** Correct handling of concurrent data access to buffers and blackboards without violating the ARINC 653 scheduling requirements with user mode scheduler is a hard task. Therefore the intra-partition schedulers are implemented in the kernel to simplify lock-wait-unlock and priority scheduling. In future versions we may design a solution that solves this issue while keeping a code in user space.

#### B. Configuration

The characteristic feature of real-time operating systems is deterministic behavior. The primary way to ensure reliable and dependable behavior is static pre-allocation of all resources – memory, CPU time, access to devices, etc. For instance, partition code is executed only during fixed time slots within the

schedule, no sooner, but no later. Memory is pre-allocated for every partition, memory image of the partition is fixed, no pages could be added or removed during runtime.

Many parameters of our operating system are configured statically and cannot be changed dynamically. These parameters are number of partitions and their memory size, number of ports, their names, sizes and directions, channels etc.

Configuration of the system is stored in xml documents. To keep the kernel minimal we got rid of the need to include xml parser to kernel: the configuration files are processed at build time. The processor generates C code where parameters are presented as either preprocessor macros (#define constants) or enum constants. The generated files are included in the build process.

### C. System partitions

Beside ordinary partitions, that interact with the kernel and the outer world through ARINC 653 APEX, the standard allows for so called *system partitions* that utilize interfaces outside the scope of APEX services, such as access to devices or network sockets. The standard doesn't specify their operations and interfaces other than constraints on time and space partitioning: system partitions are subject to scheduling. The difference between system partitions and kernel modules is that system partitions run in user space and have time and space partitioning constraints.

Our OS supports system partitions. From the kernel point of view system partitions are like ordinary partitions with some additional memory mapping and additional system calls. Communication between application partitions and system partitions is performed through ARINC-653 ports.

Currently we have only one system partition: the IO partition that is responsible for communication over the network. In the future we will implement a number of other system partitions – file system, graphics server,

**IO partition** has access (by corresponding entry in TLB) to special memory areas, where network card registers are mapped, so IO partition can work directly with hardware without kernel system calls.

IO partition receive and send data either from partitions in the same integrated module by ports or from other integrated modules by network card drivers. In the simple case the communication over network is based on UDP messages, and the configuration defines mapping between ARINC 653 port and a pair of IP address and UDP port. This mapping looks like ARINC channel, so we also call it channel.

But network communication may be based on other protocols, such as AFDX. So in general, the channel maps ARINC port to some network specific data. We support parallel work with several network protocols, by assigning channel driver to channel. Channel driver is interlayer between port and device driver. In most cases channel driver is a network stack.

System can have several network cards, so we support parallel independent work of several device drivers. Currently we support three network cards drivers: virtio, ne2k family and hardware cards on the platform with P3041 processor.

Each network driver manages one or more uniform devices. During initialization each driver, which cards are connected through PCI bus, registers as PCI device in PCI driver. After initialization of all network drivers PCI driver starts enumeration of PCI bus. If it finds a physical device that matches a registered PCI device, then it signals to the corresponding network driver. Network driver dynamically for every signal registers a *network device*. Network device has a name and method to send and receive data from assigned physical device. Names to network device are assigned dynamically, name is concatenation of drivers name and sequential number of current device in driver.

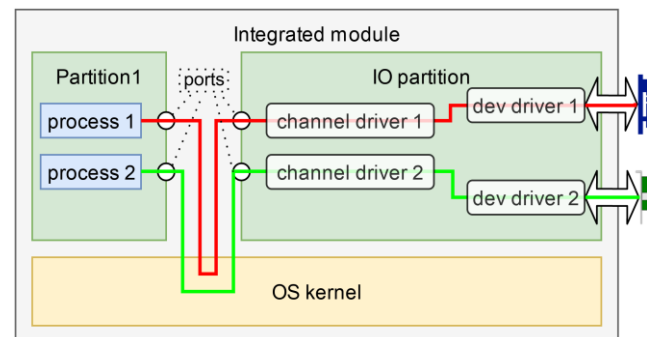


Fig. 4. Two messages are being parallel sent to different network cards

The configuration assigns channel drivers to network devices by name. Example of sending two messages in parallel to two different network cards can be seen at Fig. 4.

Different drivers require different configuration. We have dedicated xml parsers of some specific part of xml document, this parser generates data specific for corresponding driver.

This architecture allows independent work of different drivers, which can possibly come from different developers. Furthermore it allows adding new drivers with minimal effort and change of common parts.

## V. FUTURE WORK

There is research group to develop OpenGL renderer and frame buffer driver for our OS. Their work will show how well we thought out architecture of IO partition.

We finally need to measure latency without providing which we cannot tell that our operating system is a real-time system.

We are going to seek way to minimize kernel code, and move code, for which it is possible, to user-space.

Currently we use only one CPU core of the e500mc multicore processor. Newest version of ARINC 653 introduces interfaces for multicore work. We are going to support multicore CPUs as well.

Another objective is to port the OS to MIPS CPU family and another PowerPC family, namely IBM PPC 440.

## VI. CONCLUSION

In this paper we sketched Jet OS, a real-time operating system, which support ARINC 653 standard. Our system is a fork of POK OS. We describe architecture of POK, architecture of our operating system and differences between them.

## REFERENCES

- [1] Avionics application software standard interface part 0 overview of ARINC 653, ARINC specification 653P0-1, August 3, 2015
- [2] Avionics application software standard interface part 1 – required services, ARINC specification 653P1-3, november 15, 2010
- [3] RTEMS <http://www.rtems.com>
- [4] FreeRTOS <http://www.freertos.org/>
- [5] VxWorks <http://www.windriver.com>

- [6] PikeOS <https://www.sysgo.com>
- [7] LynxOS <http://www.linuxworks.com/rtos/>
- [8] Masmano, M., Valiente, Y., Balbastre, P., Ripoll, I., Crespo, A. and Metge, J.J., 2010. LithOS: a ARINC-653 guest operating for XtratuM. In Proc. of the 12th Real-Time Linux Workshop, Nairobi (Kenya).
- [9] XtratuM <http://www.xtratum.org>
- [10] S. H. VanderLeest. ARINC 653 hypervisor. In Proc. Of IEEE/AIAA DASC, Oct. 2010.
- [11] Delange, J. and Lec, L., 2011. POK, an ARINC653-compliant operating system released under the BSD license. In 13th Real-Time Linux Workshop (Vol. 10).
- [12] Sanghyun Han and Hyun-Wook Jin. 2012. Kernel-level ARINC 653 partitioning for Linux. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, 1632-1637. DOI=<http://dx.doi.org/10.1145/2245276.2232037>
- [13] J.Delange, L.Lec. POK, an ARINC653-compliant operating system released under the BSD license, 2011, <http://julien.gunnm.org/data/publications/articled111-osad111.pdf>



# Developing a Debugger for Real-Time Operating System

Alexander Emelenko

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
[emelenko@ispras.ru](mailto:emelenko@ispras.ru)

Kurban Mallachiev

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
[mallachiev@ispras.ru](mailto:mallachiev@ispras.ru)

Nikolay Pakulin

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
[npak@ispras.ru](mailto:npak@ispras.ru)

**Abstract**—In this paper we report on the work in progress on the debugger project for real-time operating system ISPRAS RTOS for civil airborne systems. We discuss the major requirements to such a debugger, review a number of debuggers for various embedded systems, and present our solution, that works both in emulator QEMU and on the target hardware. The presented debugger is based on GDB debugging framework but contains a number of extensions specific for debugging embedded applications.

**Keywords**—*debugger; GDB; real-time OS; remote debugger*

## I. INTRODUCTION

Application debugger is an indispensable tool in developer's hands. But debugger in a real-time operating system is more than just plain debugger. In this paper we present an on-going project on debugger development for ISPRAS RTOS, a real-time operating system that is being developed in the Institute for System Programming of the Russian Academy of Sciences.

ISPRAS RTOS is a prototype operating system for civil airborne avionics. It is designed to work within Integrated Modular Avionics (IMA) architecture and implements ARINC-653 API specification, the de-facto architecture for applied (functional) software.

The primary objectives of ARINC 653 are deterministic behavior and reliable execution of the functional software. To achieve this ARINC-653 imposes strict requirements on time and space partitioning. For instance, all memory allocations and execution schedules are pre-defined statically.

The unit of partitioning in ARINC-653 is called *partition*. Every partition has its own memory space and is executed in user mode. Partitions consist of one or more *processes*, operating concurrently, that share the same address space. Processes have data and stack areas and they resemble well-known concept of threads.

Embedded applications might be run in two different environments: in an emulator and on the target hardware. In our project we use QEMU system emulator. Although QEMU has its own debugger support, its functionality proved to be insufficient for debugging embedded applications. Therefore

we implemented a debugger not only for the target hardware, but for the emulator as well.

## II. MAIN TARGETS FOR DEBUGGER

Debugger for an embedded operating system has a number of specific features compared to typical debugger used by desktop developers.

Firstly, an embedded application runs under constrained conditions, such as limited on-board resources and lack of interactive facilities – no keyboard and screen. This makes it impossible to do debugging on the same device where application runs. Therefore the debugger for embedded applications has to be remote: the developer interacts with workstation while the application runs on a target hardware.

Secondly, an embedded application typically consists of a number of interacting processes that needs to be debugged simultaneously. This means that the debugger must support dynamic and transparent switching between execution contexts during debugging session.

Thirdly, the debugged should support developers of system software, mostly device drivers and network stack. This requires switching between low-privilege code and highly privileged kernel code in the same debugging.

It is also important to mention that embedded developers widely use emulators in their work process. Typically most of development runs on top of emulators, therefore the debugger must support corresponding emulators as well.

The above mentioned features impose a number of restrictions on the design of the debugger that we considered:

- There are many different applications compiled in OS, which can have overlapping virtual address spaces.
- Typically target hardware board for embedded OS has only one port to communicate with the external world – a single serial port. Since it is used to stream console output of the running applications we need to share it between debugger traffic and applications' output.

- Multifunctional debugger is a complex program. It is very complicated to develop it from scratch, so we decided to base our debugger on an existing one.
- Support debugging both on hardware and with emulator because this support can expand developers' capabilities and improve their efficiency.
- Support capabilities of debugging for kernel and for user mode code, as well as capabilities of multiprocess mode.
- It must excel QEMU debugger, which we use to emulate environment for our system.
- Since the OS in question is real-time, it is important to minimize debugger's impact on system during debugging.

In order to meet these restrictions we selected the architecture of remote debugger with server and client parts, that communicate over a serial port using multiplexer.

We have chosen GDB (GNU debugger) for the client part of our debugger.

### III. RELATED WORKS

We are not the first to consider the problem of remote debugging. For example, Pistachio microkernel uses kdebug for debugging [4]; besides, there is Fiasco debugger [1] and many different debuggers for VxWorks, for example, RTOS debugger [2].

Here we briefly consider some debuggers for embedded OSes and their primary features.

#### A. Fiasco OS

Fiasco OS is a 3rd-generation microkernel, based on L4 microkernel [1]. The kernel is simplistic, it misses most of the features available in "big" operating systems like Linux or Windows: program loading, device drivers and file system. All these features must be implemented in user-level programs on top of it (L4 Runtime Environment provides a basic set such functions).

Fiasco OS has built-in support for debugger that:

- supports threads;
- provides stack backtrace
- sets breakpoints;
- does single step;
- provides reading/writing in memory;
- provides reading hardware registers;
- support interprocess communication (IPC) monitoring.

The Fiasco Kernel Debugger (JDB) is a debugger for Fiasco. It has the following special functionality:

- It always freezes the system when it is working. It means that JDB disables all interrupts and halts clock.

All processes and kernel don't work when JDB is invoked.

- JDB doesn't use any part of Fiasco kernel, because it is a stand-alone debugger with drivers for keyboard, display, etc.

In general, JDB is not a part of Fiasco  $\mu$ -kernel, and Fiasco  $\mu$ -kernel can run without connection with JDB or another debugger.

The debugger operates remotely over the serial line.

#### B. VxWorks

VxWorks [5] is a real-time operating system (RTOS) developed as proprietary software by Wind River of Alameda, California, US. It supports Intel (x86, including the new Intel Quark SoC and x86-64), MIPS, PowerPC, SH-4, and ARM architectures.

RTOS debugger for VxWorks implements the following set of features:

- Task Stack Coverage
- Task Related Breakpoints
- Task Context Display
- Debugging Modules (for example, Kernel module)
- Debugging Real-Time Processes
- Debugging Protection Domains
- Collecting statistics for function and tasks

RTOS debugger displays all system states, tasks, message queues, memory partitioning, modules and etc.

The key feature of the RTOS debugger is that is based on Lauterbach's TRACE32 debugger [3] that utilizes hardware interfaces like JTAG. It does not use serial port for communication with the target hardware but rather requires specific debug module.

#### C. L4Ka::Pistachio:

L4Ka::Pistachio [4] is the latest L4 microkernel developed by the System Architecture Group at the University of Karlsruhe. It is the first available kernel implementation of the L4 Version 4 kernel API, which provides support of both 32-bit and 64-bit architectures, multiprocessing and superfast local IPC. The current release supports x86-x64 (AMD64/ EM64T, K9 / P4 and higher), x86-x32 (IA32, Pentium and higher), PowerPC 32bit (IBM 440, AMCC Ebony / Blue Gene P).

The debugger for Pistachio kernel can direct its I/O via the serial line or the keyboard/screen. It is a local debugger and does not support remote debugging mode.

This debugger is also a low-level device with very limited amount of functions.

Debugger for Pistachio can:

- Set breakpoints

- Single step
- Dump memory
- Read registers

When the processor meets special instruction (for example, *int3* instruction), it passes control to interrupt handler, which is the part of Pistachio kernel. In turn, interrupt handler checks instructions, which come next, and if they correspond to the special layout, it prints special message before passing control to interrupt handler. This feature is a simplistic implementation of a facility to trace execution.

#### IV. TECHNICAL DESCRIPTION:

The primary goal of the debugger is Power PC platform, based on e500mc CPU core. The debugger is based on GDB, it uses the GDB architecture to establish link to the remote target.

The architecture includes three major components: front end, local client and remote server. The front end provides user interface, it runs on the same workstation as the client part. The latter translates the commands from the front end into GDB protocol and communicates with the remote server. The server implements the actual command embedded into protocol messages such as reading memory regions, setting breakpoints, processing debug interrupts, etc. Remote server is sometimes called “stub”.

Gdb-stub for i386 was taken as a basis for our debugger. This stub was totally redesigned for e500mc processor, which belongs to PowerPC architecture family. We left only the packet exchange and some of the packet processing mechanisms.

We use common gdb client, which was built for PowerPC with somewhat extended functional, to connect to our stub. This functional was developed using special user defines commands, so developers don't need to use special version of GDB. Instead, they can use any version, but it needs to use gdb commands file by utilizing special “source” command in GDB.

Accordingly, messaging mechanism between client and server doesn't change – the client sends a special-type packet to the server and waits for the server's answer. The server receives this message, checks control sum, which was sent in this packet, and if it matches the message contents, informs the client that the message was accepted for processing. Then the server performs the action described in the packet and sends its own packet to the client.

Let us consider an example on Fig. 1. Here client sends to server packet “\$m8000acac,4#1d”. This means that client wants to read 4 bytes of memory from virtual address 0x8000acac. In this packet “1d” is the control sum, that is, the sum of all bytes in message modulo 256. If the server fully receives this message, it sends “+”, and the client knows that the message was accepted. After that, server sends 4 bytes of memory from that address to the client in the same way, and message exchange continues. All these types of packets are described in GDB manual.

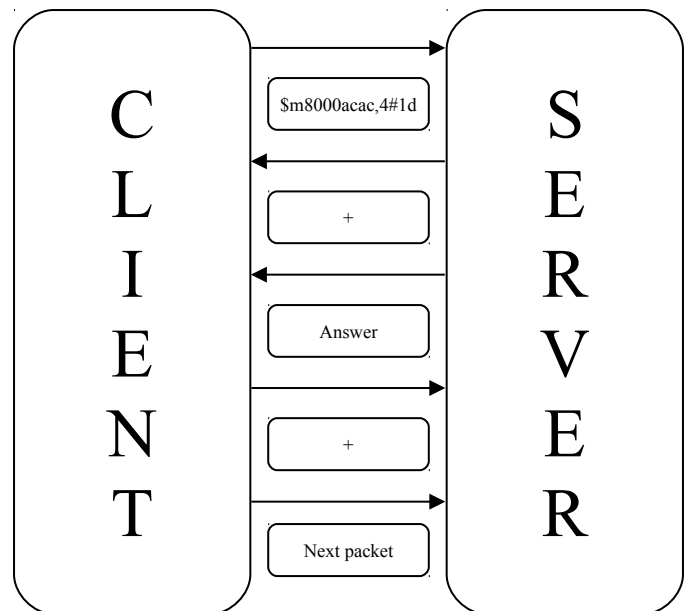


Fig. 1. GDB messaging mechanism

#### Implementation of the server side

In general, debugger's work consists of packet exchange between client and server. Client sends certain types of packets to perform the action, which the user needs. Our goal is to develop server part because we use client part from common GDB.

During the connection between server part of the debugger with client part our system stands in frozen state where no interrupts are available and the clock is halted. This opportunity allows us to work with partitions and debugger as if there is no debugger in the system.

We implemented functions in our debugger in the following way:

Breakpoint setting was implemented using special PowerPC instruction 'trap'. When the trap instruction occurs, server code in interrupt handler is called.

For Single step operation, we can use two different methods. The first one is when the system stops on the next instruction of the current partition. The second one is to stop the system stops on the next instruction wherever it is. The difference is how system calls are handled; the first method skips all kernel code and traverses application only. The second method allows entering kernel and stepping through system call implementation. Furthermore, it is sensitive to interrupts: if an interrupt occurs during the step, the debugger switched to the interrupt handler.

However, GDB structure requires interrupts to be disabled during single step. This requirement imposes restrictions on partition's work, so we gave up the second method. Because of the lack of debug registers in QEMU we need to disable interrupts and set trap instruction on the next instruction.

Watchpoints were implemented using special capabilities of hardware, such as Debug registers. Unfortunately, QEMU doesn't have such registers, so we need to use another way to



set watchpoints in emulator. This method isn't implemented yet, but we are working in this direction.

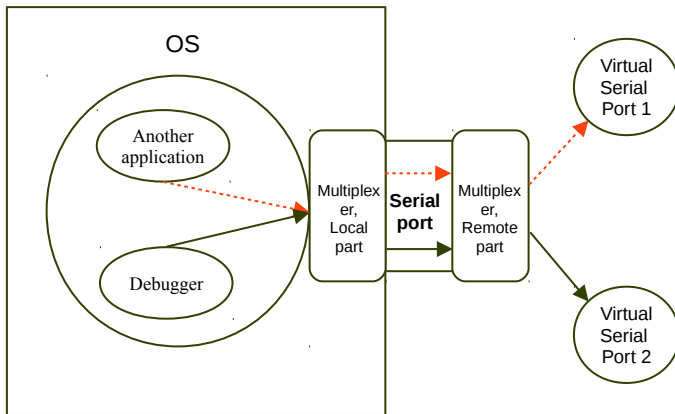


Fig. 2. Multiplexer work

We also developed multiplexer to use one serial port for both GDB and another application. Multiplexer allows message exchange for debugger and for internal system service. The transformation of one serial port into two serial ports with the help of our multiplexer is not so difficult.

There are two parts of multiplexer, local and remote. Local part is a superstructure responsible for information input/output in the system. During the output it puts a special symbol before every printable symbol, determining to which of the two virtual serial ports the next symbol should be sent. Working with input symbols is very similar: two symbols are read, with the first of them specifying the application to which we want to send the second symbol. Remote part of multiplexer looks the same. This solution is not the fastest, but it provides smooth debugger's work via one serial port together with other applications. This connection between remote and local parts of multiplexer is shown on Fig. 2.

## V. DEBUGGER'S CAPABILITIES

Our debugger supports all standard debugging features. Among them are:

### 1. Setting Breakpoints on Kernel and Partitions.

Setting breakpoints is the key feature of any debugger. Considering that client knows only virtual addresses, the server part of the debugger must correctly translate this address into physical address. Our debugger can do this, that's why users can debug partitions with overlapping virtual address spaces and debugger stops only on the partition that the user wants.

### 2. Single Step.

Stepping through code step by step is a convenient way of finding bugs. However, there can be a situation in real-time OS, when the next instruction in code is not the next executable instruction, for example, because of timer interrupt. That's why we disable interrupts during the single step.

### 3. Showing Information about Processes and Threads, Inspecting Memory, Instructions and Registers. Memory Reading and Writing.

Memory view must correctly translate virtual addresses into physical as with breakpoints. The capability to find out all information about threads in OS, their states, registers and memory is very important too.

Support of memory writes allows changing process state as user's discretion.

### 4. Setting Watchpoints.

Watchpoints are one of the most comfortable ways to control user's partition. They give the opportunity to follow changes in memory sectors and stop/pause while trying to read or record memory. This opportunity increases the number of ways to control partitions' states.

### 5. Stack Inspection.

Stack inspection makes tracing possible: for example, tracing the queue of called functions, which can help user to understand exactly what has happened in the system.

## VI. FUTURE WORK

Implementation of the debugger is not complete yet. There are a number of features that can improve debugger usability:

- Enhance debugging capabilities to the level of standard GDB functionality.
- Accelerate debugger interaction time with the system through multiplexer.
- Improve hardware support on bare metal.
- Increase user convenience in multiplexer. Enhance its functionality for working with more devices (now multiplexer supports only two devices). This solution allows us to work on bare metal with as many ports as we need, regardless of the actual amount of ports.
- Add watchpoints implementation to QEMU, which doesn't support debug registers. This is the reason why we can't use debug registers for setting watchpoints like we do on bare metal. In that case, we need to change code handling in QEMU to develop instruction for watchpoints creation.

## VII. CONCLUSION

In this paper we have presented our project on implementation of the debugger for real-time operating system ISPRAS RTOS. In contrast to other systems and their debuggers, where user can use some functions to debug applications, but not all we need, our debugger meets the majority of our requirements and restrictions. However, we will be able to update our debugger in near future and increase its functionality, but it is already more functional than common GDB debugger for QEMU.

## REFERENCES

- [1] F. Mehnert, J. Glauber and J. Liedtke, “Fiasco Kernel Debugger Manual” Dresden University of Technology, Department of Computer Science, November 2008
- [2] Lauterbach GmbH, “RTOS debugger for VxWorks”, November 2015
- [3] Lauterbach GmbH, “RTOS-VxWorks”, 18 August 2014
- [4] System Architecture Group University of Karlsruhe. “The L4Ka::Pistachio Microkernel”. May 1, 2003
- [5] Wind River Systems, Inc “VxWorks Product Overview”, March 2016
- [6] Free Software Foundation, Inc. “Debugging with gdb: the gnu Source-Level Debugger”, The Tenth Edition

# Building and Testing an Embedded Operating System

Alexey Ovcharov

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
aovch@ispras.ru

Nikolay Pakulin

Institute for System Programming  
of the Russian Academy of Sciences  
Moscow, Russian Federation  
npak@ispras.ru

**Abstract**—In this paper we report on the work in progress concerning embedded OS building and testing environment. By switching from make to SCons, which is essentially Python, we achieve a greater level of convenience and maintainability.

**Keywords**—*embedded OS; SCons; software development*

## I. INTRODUCTION

It is only natural to wish for simplicity instead of complexity. Simplicity is next to elegance, and complexity is error-prone. In the world of rapidly evolving software, where a typical project codebase is measured in thousands or even millions of source lines of code, maintainability is an issue of highest priority. A project needs to be reconfigured, rebuilt, and put to tests countless times a day. It goes without saying, that maintenance overhead has the annoying ability to stall the development.

Working on a real-time operating system, we were faced with a challenge of making our build and test system as simple and convenient as possible. Traditional Makefiles, used in the project when we set to work on the OS, did not provide the required flexibility and ease of maintenance. Makefiles were difficult to read, dependencies between files were handled badly, and every change in the source, even the smallest one, led to the full rebuild. It was a matter of utmost importance, and it needed to be resolved quickly.

Of course, we could honor the traditions and cling to Make. We could rewrite the Makefiles (from scratch, apparently). Instead, we opted for SCons [1] (mostly because it was used in our previous project and we already saw its potential).

The OS we are working on is developed in pure C and ASM. It is quite small in size, totaling in about 30,000 lines of code. It consists of nothing more than the kernel, several BSP, a single userspace library, and a couple of user applications.

As befits an embedded OS, our OS cannot be changed (e.g., extended with new applications) when in operation. The majority of parameters such as the number of processes and ports in the system are configured statically, and the kernel and user applications are linked into a single bootable image.

Hence, every application-like test (of which we have approximately 400) must be a separate complete OS image. Nevertheless, when the test suite is run, the kernel can be reused for every test, so only test files and bootable images

must be built. This was also taken into account when rewriting our build system.

Our first and foremost goal was to reduce the impact of modifications in the source code on build time. We did not want to rebuild the whole OS when all that changed was a single source file, and we wanted the dependencies to be handled correctly when, for example, a header file was modified. In the following section we will discuss several mature OS with regard to their maintenance systems.

## II. KNOWN APPROACHES TO EMBEDDED OS BUILDING

Our OS is not the first and, probably, not the last developed embedded OS. In this section we will briefly review OS such as VxWorks [2], PikeOS [3], and RTEMS [4].

VxWorks is a safety-critical OS with modular architecture. Wind River Workbench, the Eclipse-based configuration and build system, is designed with three goals in mind. The first goal is the dependency minimization, which makes it possible to build and certify different elements of the module separately. The second goal is the support of incremental building in order to reduce the number of files that have to be rebuilt as a consequence of source change. The third goal is the distributed development, allowing to smoothly divide tasks between different development groups.

VxWorks modules can be built both with command-line tools and through the GUI [2]. The build system is based on Makefiles and provides the user with the control over three aspects of the system: mode (cert or debug), CPU for which to build the module, and image type (RAM, ROM, or network-loadable).

PikeOS is bundled with CODEO, the Eclipse-based IDE [3]. Among its facilities are configuration editor, integrity checker, system monitor, and so on.

In contrast to proprietary VxWorks and PikeOS, RTEMS is an open-source OS. RTEMS supports development with Makefiles as well as with Eclipse. For integration with Eclipse, a CDT-based plugin is provided [4].

## III. SCONS

SCons is a Python-based software construction tool [1]. SCons is better than Make for a variety of reasons. First of all, configuration scripts are written in Python. The ability to use a full-fledged programming language is a blessing by itself;

however, SCons offers a range of other features, each of them a reason to switch to this instrument.

SCons uses MD5 signatures to detect changes in the source tree. This is best coupled with caching of built files to speed up the builds. Fortunately, SCons can cache all types of files, while ccache works only with C/C++ compilation output.

Another advantage of SCons over Make is sophisticated dependency handling. SCons does not require a separate launch to generate dependencies – they are deduced automatically. In our case, we need to generate C source files from user application XML configuration before building the system. These C files are specified as source for the application ELF file, and modifications in XML are transferred to corresponding ELFs. Of course, if the changes in XML do not impact the resulting C files (e.g., a comment was added), no rebuild will be necessary.

If out-of-the-box dependency handling turns out to be not enough, additional dependencies can easily be defined in the building scripts. For example, our ELFs depend on BSP-specific linker scripts, which default SCons builders cannot handle properly. There is a simple solution to this problem, however. We just had to add “-T /path/to/linker/script.lds” string to linker flags and make the resulting ELF depend on the linker script with the call to Depends method.

Naturally, SCons supports parallel builds, as does Make.

Cleaning the project is as simple as typing “scons -c”, and everything created during the build is located via the dependency tree and removed. Should the need arise, SCons provides the ability to define *clean* targets similar to those of Make.

Unfortunately, SCons does not come without drawbacks. When launched, the utility first scans all its scripts in the project in order to construct a dependency tree. This can take some time, and might be a bit frustrating, especially if SCons decides that there is nothing to build. In our case, an idle run on an already built project takes SCons slightly less than two seconds. Nevertheless, we consider it a step forward from the full rebuild.

Another major drawback of migrating from Make to SCons is the need to rewrite the build system, but once it is done, adding files to build and tweaking the scripts becomes a lot more comfortable.

#### IV. VARIOUS ISSUES

This section will be dedicated to the variety of problems we had to take into account when working on the build system.

##### 1. *Building for multiple BSP.*

An embedded OS targeted at a single BSP does not stand a chance against its more broad-minded counterparts. The wider the BSP range, the higher the OS chances of attracting customers. But how to organize support for multiple BSP?

SCons provides VariantDir method, allowing to set up multiple builds with different options and separate built files from source.

Naturally, each architecture requires a different toolchain. To specify BSP, we pass it to SCons as a command-line parameter. The set of tools suitable for a particular architecture is defined in one of our SCons scripts, and can be effortlessly extended should a new architecture come into play.

##### 2. *Distributed development.*

It would be very unwise to deny application developers the ability to work independently from each other. Our build system enables the pre-built applications to be configured by system integrator and linked into a single bootable OS image.

##### 3. *Running and debugging.*

Our OS runs under QEMU, an open-source machine emulator. To boot the OS image, QEMU requires a number of BSP-specific parameters. For the sake of convenience, we construct the suitable command line in SCons scripts and define target “run” as an alias to it. Thus, “scons run” executes this command, launching QEMU with the OS image.

QEMU is equipped with gdbserver to facilitate emulated targets debugging via gdb. It is enabled by yet another option, so we define two more targets: “rundbg” for launching QEMU ready for connection from gdb, and “debug” for gdb itself. To debug the OS with an application, users must open two terminals in this application's directory, then type “scons rundbg” in one terminal and “scons debug” in the other.

#### V. FUTURE WORK

There is no such thing as too many test runs. It seems like a good idea to add nightly build functionality to our project. We are currently looking into buildbot in the hope that it will help us automate scheduled testing.

#### VI. CONCLUSION

In this paper we have presented our improvements to OS building/testing environment. Giving up Make in favor of SCons, we gain a significant boost in convenience and maintainability. Whereas with Make our test suite completed in a several hours, with SCons it now takes about 20 minutes.

#### REFERENCES

- [1] SCons: A software construction tool [Online]. Available: <http://scons.org/>
- [2] Wind River. VxWorks 653 Configuration and Build Guide 2.3 Edition 2.
- [3] PikeOS Hypervisor: Eclipse based CODEO [Online]. Available: <https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/eclipse-based-codeo/>
- [4] RTEMS Eclipse Support [Online]. Available: <https://dev.rtems.org/wiki/Developer/Eclipse/Information>